Appunti dal corso di calcolo numerico

Carli Samuele
Matricola: 4036768
E-mail: winsucks@tin.it

Gennaio 2007

Indice

1 Errori ed aritmetica finita **7**
- 1.1 Valutazione dell'errore . . . 7
  - 1.1.1 L'errore assoluto . . . 7
  - 1.1.2 L'errore relativo . . . 7
- 1.2 Sorgenti di errore . . . 8
  - 1.2.1 Errori di discretizzazione . . . 8
  - 1.2.2 Errori di convergenza . . . 8
  - 1.2.3 Errori di *round-off* . . . 10
- 1.3 Rappresentazione dei numeri reali . . . 11
  - 1.3.1 I numeri di macchina . . . 11
  - 1.3.2 Overflow e underflow . . . 13
- 1.4 Lo standard IEEE 754 . . . 14
- 1.5 Condizionamento di un problema . . . 15
  - 1.5.1 Condizionamento delle operazioni elementari . . . 16

2 Radici di una equazione **19**
- 2.1 Il metodo di bisezione . . . 19
  - 2.1.1 Criteri di arresto: . . . 20
  - 2.1.2 Ordine di convergenza . . . 22
- 2.2 Il metodo di Newton . . . 22
  - 2.2.1 Convergenza locale . . . 24
  - 2.2.2 Criteri di arresto . . . 25
  - 2.2.3 Ripristino della convergenza quadratica in caso di radici multiple 28
- 2.3 Metodi quasi-Newton . . . 34
  - 2.3.1 Metodo delle corde . . . 34
  - 2.3.2 Metodo delle secanti . . . 35

3 Risoluzione di sistemi lineari **41**
- 3.1 Matrici quadrate . . . 41
  - 3.1.1 Matrici diagonali . . . 41
  - 3.1.2 Matrici triangolari . . . 42
  - 3.1.3 Matrici ortogonali . . . 43
  - 3.1.4 Proprietà . . . 43
- 3.2 Metodi di fattorizzazione . . . 46
- 3.3 Fattorizzazione LU . . . 46
  - 3.3.1 Costo computazionale . . . 50
- 3.4 Matrici a diagonale dominante . . . 51
- 3.5 Fattorizzazione LDL^T . . . 52
  - 3.5.1 Costo computazionale . . . 54

3.6 Pivoting . . . 56
3.7 Condizionamento del problema . . . 59
3.8 Sistemi lineari sovradeterminati . . . 62
3.8.1 Esistenza della fattorizzazione QR . . . 63

4 Approssimazione di funzioni **69**
4.1 Interpolazione polinomiale . . . 69
4.2 Forma di Lagrange e forma di Newton . . . 70
4.3 Interpolazione di Hermite . . . 74
4.4 Errore nell'interpolazione . . . 77
4.5 Condizionamento del problema . . . 80
4.6 Ascisse di Chebyshev . . . 81
4.7 Funzioni *spline* . . . 83
4.7.1 Spline cubiche . . . 85
4.7.2 Calcolo di una spline cubica . . . 86
4.8 Approssimazione polinomiale ai minimi quadrati . . . 90

5 Formule di quadratura **99**
5.1 Formule di Newton-Cotes . . . 99
5.1.1 Formula dei trapezi . . . 100
5.1.2 Formula di Simpson . . . 100
5.1.3 Condizionamento del problema . . . 102
5.2 Errore e formule composite . . . 102
5.2.1 Formule composite . . . 103
5.3 Formule adattative . . . 104
5.3.1 Formula dei trapezi . . . 105
5.3.2 Formula di Simpson . . . 105

6 Altre implementazioni **109**
6.1 Modifica al metodo delle secanti . . . 109
6.2 Fattorizzazione LU . . . 111
6.2.1 Algoritmo ottimizzato . . . 111
6.2.2 Ottenere i fattori L ed U . . . 112
6.2.3 Risolvere il sistema . . . 112
6.2.4 Ottimizzazione per sistemi tridiagonali . . . 113
6.3 Fattorizzazione LDL^T . . . 114
6.3.1 Algoritmo ottimizzato . . . 114
6.3.2 Trasformare una matrice sdp in un vettore utile all'utilizzo dell'algoritmo ottimizzato . . . 115
6.3.3 Piccolo miglioramento . . . 116
6.3.4 Ottenere i fattori L e D . . . 119
6.3.5 Risolvere il sistema . . . 122
6.4 Fattorizzazione LU con pivoting parziale . . . 122
6.4.1 Algoritmo ottimizzato . . . 122
6.4.2 Risolvere il sistema . . . 123
6.5 Fattorizzazione QR . . . 124
6.5.1 Ottenere i fattori Q' ed R . . . 124
6.5.2 Risolvere il sistema . . . 125
6.6 Algoritmo di Horner . . . 126
6.6.1 Algoritmo originale . . . 126
6.6.2 Algoritmo greneralizzato . . . 126
6.6.3 Polinomio interpolante di Newton . . . 126
6.6.4 Polinomio interpolante di Hermite . . . 127
6.7 Spline . . . 127

6.7.1 Spline lineare 127
6.7.2 Spline cubica naturale 128
6.7.3 Spline cubica not-a-knot 130
6.7.4 Spline cubica completa 133
6.7.5 Spline cubica periodica 135
6.7.6 Approssimazione ai mini quadrati 137
6.8 Formule di quadratura 138
6.8.1 Formula dei trapezi adattiva non ricorsiva 138
6.8.2 Formula di Simpson adattiva non ricorsiva 140

Capitolo 1

Errori ed aritmetica finita

1.1 Valutazione dell'errore

1.1.1 L'errore assoluto

Si supponga di avere a disposizione un metodo numerico che dia come risultato un numero $\tilde{x} \in \mathbb{R}$ approssimazione del risultato *esatto* $x \in \mathbb{R}$. É possibile definire la grandezza **errore assoluto** come:

$$\triangle x = \tilde{x} - x$$

o in altra forma:

$$\tilde{x} = x + \triangle x.$$

Questa grandezza, benché fornisca un'indicazione precisa del valore dell'errore commesso dal metodo matematico preso in esame, non permette di valutare quanta influenza abbia l'errore sul risultato ottenuto. Ad esempio, un errore assoluto $\triangle x = O(10^{-5})$ potrebbe avere un peso accettabile se il nostro risultato corretto x fosse dell'ordine di grandezza $O(1)$ o superiore, ma non si avrebbe un risultato valutabile se x fosse anch'esso $O(10^{-5})$.

1.1.2 L'errore relativo

Volendo conoscere quanto un errore influenzi il risultato quando $x \neq 0$, si definisce l'errore relativo:

$$\varepsilon_x \equiv \frac{\triangle x}{x} = \frac{\tilde{x} - x}{x}$$

da cui:

$$\tilde{x} = x(1 + \varepsilon_x), \quad \text{e quindi} \quad \frac{\tilde{x}}{x} = 1 + \varepsilon_x$$

ovvero l'errore relativo deve essere comparato a 1: un errore relativo vicino a zero indicherà che il risultato approssimato è molto vicino al risultato esatto,mentre un errore relativo uguale a 1 indicherà la totale perdita di informazione.

Esempi

Sia $x = \pi \simeq 3.1415 = \tilde{x}$. L'errore relativo è quindi: $\varepsilon_x = \frac{\tilde{x}-x}{x} \simeq \frac{3.1415-3.141592653}{3.141592653} = -0.000029492$. Notiamo che la formula: $-log_{10}(\varepsilon_x)$ restituisce all'incirca il numero di cifre significative corrette all'interno di $\tilde{x}$: in questo caso il risultato del calcolo è 4.53, che è abbastanza vicino alla realtà di 5 cifre significative corrette.

1.2 Sorgenti di errore

1.2.1 Errori di discretizzazione

I calcolatori elettronici non sono in grado di lavorare su un insieme numerico continuo, ma sono limitati a operare su un insieme numerico finito e discreto. In generale, però, i problemi matematici sono naturalmente pensati su un insieme continuo, ed è necessario sostituirli con opportuni problemi discreti che il calcolatore è in grado di gestire. Ad esempio, si supponga di voler calcolare un'approssimazione della derivata prima di una funzione derivabile f in un punto assegnato x_0:

$$f'(x_0) \simeq \frac{f(x_0+h)-f(x_0)}{h} \text{ con } h>0.$$

L'incremento h, per quanto piccolo, è una quantità finita.
Consideriamo l'espansione in serie di Taylor con resto al secondo ordine:

$$f(x_0+h) = f(x_0) + hf'(x_0) + \tfrac{h^2}{2!}f''(\psi) \text{ con } \psi \in (x_0, x_0+h)$$

sostituendo si ha:

$$f'(x_0) \simeq \frac{f(x_0+h)-f(x_0)}{h} = f'(x_0) + \tfrac{h}{2}f''(\psi)$$

dove il termine $\frac{h}{2}f''(\psi)$ quantifica l'errore di discretizzazione, che in questo caso è un $O(h)$.
Consideriamo invece la formula:

$$f'(x_0) = \frac{f(x_0+h)-f(x_0-h)}{2h} \text{ con } h>0$$

e lo sviluppo in serie di Taylor di $f(x)$ di punto inziale x_0 con resto al terzo ordine si ottiene:

$$f'(x_0) = \frac{(f(x_0)+hf'(x_0)+\frac{h^2}{2!}f''(x_0)+\frac{h^3}{3!}f'''(\psi)) - (f(x_0)-hf'(x_0)+\frac{h^2}{2!}f''(x_0)-\frac{h^3}{3!}f'''(\eta))}{2h}$$
$$\text{con } \psi \in (x_0, x_0+h) \text{ e } \eta \in (x_0-h, x_0)$$

ovvero:

$$f'(x_0) = f'(x_0) + \tfrac{h^2}{12}(f'''(\psi) - f'''(\eta))$$

In questo caso l'errore commesso è $O(h^2)$, minore rispetto al caso precedente.
Questo esempio mostra come, approssimando un problema continuo a un problema discreto in due modi diversi, vi possa essere una grossa differenza nell'errore di discretizzazione commesso.

1.2.2 Errori di convergenza

Spesso per risolvere un problema è possibile definire un metodo numerico di tipo *iterativo* che approssimi il risultato *esatto*, qui indicato con $\bar{x}$, attraverso una successione di risultati intermedi $\{x_n\}$, ricavabili mediante la *funzione di iterazione del metodo*:

$$x_{n+1} = \Phi(x_n), \quad n = 0, 1, 2...,$$

Ovviamente, affinché il calcolo iterativo possa avere inizio, è necessario definire un x_0 *approssimazione iniziale* su cui eseguire il metodo la prima volta.
Condizione minimale che rende il metodo iterativo utile è che sia verificata la *condizione di consistenza*: $\Phi(\bar{x}) = \bar{x}$.

Un metodo iterativo si potrà dire convergente se:

$$\lim_{n\to\infty} x_n = \bar{x}$$

tuttavia, il risultato corretto è disponibile solo dopo infinite iterazioni. Ovviamente al di fuori del mondo matematico si sarà costretti ad eseguire un numero *finito* di iterazioni, rendendo quindi necessario definire un opportuno criterio di arresto che all'n-esimo ciclo arresti la successione e determini l'errore:

$$\triangle x = x_n - \bar{x}$$

detto *errore assoluto di convergenza* del metodo $\Phi(x_n)$.

Teorema

Il metodo iterativo appena descritto, convergente a $\bar{x}$, deve verificare la condizione di consistenza; ovvero la soluzione cercata deve essere un punto fisso per la funzione di iterazione che definisce il metodo.
Dimostrazione:
Supponiamo che non valga la condizione di consistenza, ovvero $\Phi(\bar{x}) \neq \bar{x}$. Allora se all'$n$-esimo passo dell'iterazione $x_n = \bar{x}$, $x_{n+1} \neq \bar{x}$ e non sarebbe vero che $\lim_{n\to\infty} x_n = \bar{x}$, ovvero la funzione non sarebbe convergente e non sarebbe possibile arrivare a un risultato nemmeno dopo infinite iterazioni.

Esempio

Il metodo iterativo

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right), \quad n = 0, 1, 2..., \quad x_0 = 2$$

definisce un metodo numerico convergente per calcolare $\sqrt{2}$.
Dimostrazione:
$\sqrt{2}$ è un punto fisso per la funzione, infatti:

$$\frac{1}{2}\left(\sqrt{2} + \frac{2}{\sqrt{2}}\right) = \frac{\sqrt{2}}{2} + \frac{1}{\sqrt{2}} = \frac{2}{\sqrt{2}} = \sqrt{2}.$$

e quindi vale la condizione di consistenza.
La funzione converge a $\sqrt{2}$ e a ogni passo dell'iterazione la distanza dal valore corretto diminuisce, ovvero:

$$\frac{1}{2}\left(x_n + \frac{2}{x_n}\right) - \sqrt{2} < \frac{1}{2}\left(x_{n+1} + \frac{2}{x_{n+1}}\right) - \sqrt{2}.$$

Dal momento che consideriamo $x_0 = 2$, la funzione si avvicina a $\sqrt{2}$ per valori positivi. Quindi:

$$\frac{1}{2}\left(x + \frac{2}{x}\right) - \sqrt{2} < \frac{1}{2}\left(\frac{\frac{1}{2}\left(x + \frac{2}{x}\right) + \frac{2}{\frac{1}{2}\left(x + \frac{2}{x}\right)}}{}\right) - \sqrt{2},$$

$$\frac{x^2+2}{2x} < \frac{1}{2}\left(\frac{x^2+2}{2x} + \frac{2}{\frac{x^2+2}{2x}}\right), \quad \frac{x^2+2}{2x} < \frac{1}{2}\left(\frac{\frac{(x^2+2)^2}{2x} + 4x}{2x\left(\frac{x^2+2}{2x}\right)}\right),$$

$$\frac{x^2+2}{2x} < \frac{\frac{(x^2+2)^2+8x^2}{2x}}{2(x^2+2)}, \quad (x^2+2)^2 < (x^2+2)^2 + 8x^2, \quad 8x^2 > 0, \quad x > 0.$$

Ovvero per ogni $x > 0$ punto di innesco della sequenza, questa converge a $\sqrt{2}$. Cvd.

Esempio

Il metodo iterativo:

$$x_{n+1} = \frac{x_n x_{n-1} + 2}{x_n x_{n-1}},\ n = 1, 2, ..., \qquad x_0 = 2, x_1 = 1.5,$$

fornisce una successione di approssimazioni convergente a $\sqrt{2}$.
Ecco un esempio di successione, arrestata a un errore di convergenza $\approx 10^{-12}$, considerando $\sqrt{2} = 1.414213562374$:

n	x_n	ε_{conv}
0	2	$4.1 * 10^{-1}$
1	1.5	$6.0 * 10^{-2}$
2	1.42	$4.1 * 10^{-3}$
3	1.4137	$3.6 * 10^{-4}$
4	1.41423	$1.2 * 10^{-5}$
5	1.41421358	$1.2 * 10^{-8}$
6	1.41421356237339	$4.3 * 10^{-13}$

1.2.3 Errori di *round-off*

Gli errori di *round-off* si verificano quando si cerca di rappresentare dei numeri che contengano intrinsecamente una quantità infinita di informazione (come ad esempio i numeri irrazionali) all'interno di una memoria dalle dimensioni finite che quindi obbliga a utilizzare un'approssimazione di questi.

Rappresentazione dei numeri interi

Un numero intero è rappresentato, all'interno di un calcolatore, come una stringa del tipo:

$$\alpha_0 \alpha_1 .... \alpha_n$$

per cui, assegnata una base $b \in \mathbb{N}$:

$$\alpha_0 \in \{+, -\}, \quad \alpha_i \in \{0, 1, ..., b-1\}, \quad i = 1, ..., N.$$

A questa stringa viene fatto corrispondere il numero intero:

$$n = \begin{cases} \sum_{i=1}^{N} \alpha_i b^{N-i}, \text{ se } \alpha_0 = +, \\ \\ \sum_{i=1}^{N} \alpha_i b^{N-i} - b^N, \text{ se } \alpha_0 = -. \end{cases}$$

In particolare nel caso di α_0 positivo si ha che:

$$0 \leq \alpha_1 b^{N-1} + ... + \alpha_N b^0 \leq (b-1)b^{N-1} + ... + (b-1)b^0 = (b-1)\sum_{i=1}^{N} b^{N-i}$$

ovvero il massimo numero positivo rappresentabile è:

$$(b-1)\sum_{i=1}^{N} b^{N-i} = (b-1)\sum_{i=0}^{N-1} b^i = (b-1)\frac{b^N - 1}{b-1} = b^N - 1$$

e quindi l'intervallo di numeri positivi rappresentabile è: $[0, b^N - 1]$
mentre quello di numeri negativi è: $[-b^N, -1]$.
Mediante questo sistema è possibile rappresentare senza errore tutti gli interi nell'insieme $\{-b^N, ..., b^N - 1\}$.

1.3 Rappresentazione dei numeri reali

Un numero reale è rappresentato come una stringa del tipo:

$$\alpha_0\alpha_1...\alpha_m\beta_1\beta_2...\beta_s$$

dove $\alpha_0 \in \{+,-\}, \alpha_i, \beta_j \in \{0,1,...,b-1\}$ con $i = 1...m$ e $j = 1...s$ ed $\alpha_1 \neq 0$ per la normalizzazione.
La stringa suddetta rappresenta il numero reale:

$$r = \pm\left(\sum_{i=1}^{m}\alpha_i b^{1-i}\right)b^{e-\nu}, e = \sum_{j=1}^{s}\beta_j b^{s-j}$$

in cui lo shift ν è fissato e chiameremo le quantità $\rho = \pm\sum_{i=1}^{n}\alpha_i b^{1-i}$ *mantissa* e $\mu = e - \nu$ *esponente* del numero reale. La presenza dello shift serve a evitare di sprecare un bit dell'esponente per discriminarne il segno.
In particolare si ha che:

$$-\nu \leq e \leq \frac{b^s-1}{b-1}(b-1) - \nu = b^s - 1 - \nu$$

ovvero $e = (b-1)\sum_{j=1}^{s} b^{s-j} - \nu$.

Teorema

Secondo la rappresentazione appena definita, $1 \leq \rho < b$.
Dimostrazione:

Il minimo numero rappresentabile si ha nel caso $\alpha_1 = 1, \alpha_2 = 0...\alpha_m = 0$: $1.\overbrace{000}^{m-1}$.
Il massimo invece si avrà nel caso $\alpha_1 = ... = \alpha_m = (b-1)$: $(b-1).\underbrace{(b-1)...(b-1)}_{m-1}$

In quest'ultimo caso si ha:

$$(b-1).\underbrace{(b-1)...(b-1)}_{m-1} = \sum_{i=1}^{m}(b-1)b^{1-i} = (b-1)\sum_{i=0}^{m-1}(b^{-1})^i = (b-1)\frac{1-b^{-m}}{1-b^{-1}} =$$

$$= (b-1)\frac{1-b^{-m}}{(b-1)b^{-1}} = b(1-b^{-m})$$

che è sicuramente minore di b in quanto $(1-b^{-m}) < 1$. Cvd.

Il valore assoluto minimo e massimo tra i numeri di macchina rappresentabili sono rispettivamente:

$$\begin{aligned} r1 &= b^{-\nu} \\ r2 &= b(1-b^{-m})b^{((b^s-1)-\nu)} = (1-b^{-m})b^{s-\nu} \end{aligned} .$$

1.3.1 I numeri di macchina

Definiamo adesso l'insieme dei numeri di macchina:

$$\mathcal{M} = \left\{ x \in \mathbb{R} \mid x = \pm\rho b^e, \rho = \sum_{i=1}^{m}\alpha_i b^{1-i}, e = \sum_{j=1}^{s}\beta_j b^{s-j} \right\} \cup \{0\}.$$

In particolare si ha che:

$$\mathcal{M} \subseteq [-r2, r1] \cup \{0\} \cup [r1, r2] = \mathcal{I}$$

ovvero i numeri di macchina appartengono a un sottoinsieme della retta reale che per definizione è un intervallo denso, quindi infinito e non numerabile.

Teorema

$\mathcal{M}$ ha un numero finito di elementi.
Infatti, $\alpha_2...\alpha_m\beta_1...\beta_s$ possono assumere b^{s+m-1} valori, α_1 ne può assumere $(b-1)$ e α_0 invece soltanto 2. Ovvero i numeri rappresentabili sono: $2(b-1)b^{s+m-1} < 2b^{s+m}$.
In particolare $|\rho| = \alpha_1\alpha_2...\alpha_m \geq \alpha_1 \geq 1$, ma anche:
$|\rho| = \alpha_1\alpha_2...\alpha_m \leq (b-1).\overbrace{(b-1)...(b-1)}^{m-1} \leq b$. Cvd.

Funzione floating

In quanto $\mathcal{M}$ ha un numero finito di elementi e $\mathcal{I}$ ne ha un numero infinito, è necessario definire una funzione $fl(x)$, detta *funzione floating*, che associ ad ogni numero reale $x \in \mathcal{I}$ un corrispondente numero di macchina:

$$fl : x \in \mathcal{I} \rightarrow fl(x) \in \mathcal{M}$$

Troncamento

Sia $x = \pm\alpha_1\alpha_2...\alpha_m\alpha_{m+1}...b^e \in \mathcal{I}$
Possiamo definire:

$$\tilde{x} = fl(x) = \pm\alpha_1\alpha_2...\alpha_m b^e.$$

Arrotondamento

Sia $x = \pm\alpha_1\alpha_2...\alpha_m\alpha_{m+1}...b^e \in \mathcal{I}$
Possiamo definire

$$\tilde{x} = fl(x) = \pm\alpha_1\alpha_2...\tilde{\alpha_m}b^e, \tilde{\alpha_m} = \begin{cases} \alpha_m, \text{ se } \alpha_{m+1} < \dfrac{b}{2} \\ \alpha_m + 1, \text{ se } \alpha_{m+1} \geq \dfrac{b}{2} \end{cases}.$$

Teorema

$\forall x \in \mathcal{I}, x \neq 0$:

$$\tilde{x} = fl(x) = x(1+\varepsilon_x), |\varepsilon_x| \leq u = \begin{cases} b^{1-m}\ Troncamento \\ \dfrac{1}{2}b^{1-m}\ Arrotondamento \end{cases}$$

dove con u indichiamo la precisione di macchina.
Dimostrazione (nel caso di troncamento):

$$|\varepsilon_x| = \frac{|x-\tilde{x}|}{|x|} = \frac{(\alpha_1\alpha_2...\alpha_m\alpha_{m+1}...)b^{e-\nu} - (\alpha_1\alpha_2...\alpha_m)b^{e-\nu}}{(\alpha_1\alpha_2...\alpha_m\alpha_{m+1}...)b^{e-\nu}} =$$

$$= \frac{0.\overbrace{000}^{m-1}\alpha_{m+1}...}{\alpha_1\alpha_2...\alpha_m\alpha_{m+1}...} \leq \frac{(\alpha_{m+1}\alpha_{m+2}...)b^{-m}}{1} \leq \frac{bb^{-m}}{1} = b^{1-m} = u$$

e quindi:

$$x \in \mathcal{I} \Rightarrow fl(x) = x(1-\varepsilon_x), |\varepsilon_x| \leq u$$

Osservazione: $-log_{10}|\varepsilon_x| \simeq$ numero di cifre corrette nella mantissa.
In particolare $-log_{10}|\varepsilon_x| \leq -log_{10}|u|$ e $m = 1 - log_{10}|u|$
La quantità u prende il nome di *precisione di macchina*

Dimostrazione:

nel caso di arrotondamento:
Adesso dobbiamo distinguere tra i due casi, $\alpha_{m+1} < \frac{b}{2}$ oppure $\alpha_{m+1} \geq \frac{b}{2}$.
Nel primo:

$$|\varepsilon_x| = \frac{|x - \tilde{x}|}{|x|} = \frac{(\alpha_1\alpha_2...\alpha_m\alpha_{m+1}...)b^{e-\nu} - (\alpha_1\alpha_2...\tilde{\alpha_m})b^{e-\nu}}{(\alpha_1\alpha_2...\alpha_m\alpha_{m+1}...)b^{e-\nu}} =$$

$$= \frac{0.\overbrace{000}^{m-1}\alpha_{m+1}...}{\alpha_1\alpha_2...\alpha_m\alpha_{m+1}...} \leq \frac{(\alpha_{m+1}\alpha_{m+2}...)b^{-m}}{1} \leq \frac{\frac{1}{2}bb^{-m}}{1} = \frac{1}{2}b^{1-m}$$

osservazione: $(\alpha_{m+1}\alpha_{m+2}...)$ è minore di $b/2$ per ipotesi.

Nel secondo:

$$|\varepsilon_x| = \frac{|x - \tilde{x}|}{|x|} = \frac{|x - \tilde{x}|}{x} = \frac{(\alpha_1\alpha_2...\alpha_m\alpha_{m+1}...)b^{e-\nu} - (\alpha_1\alpha_2...\tilde{\alpha_m})b^{e-\nu}}{(\alpha_1\alpha_2...\alpha_m\alpha_{m+1}...)b^{e-\nu}} =$$

$$= \frac{|-0.\overbrace{000}^{m-1}\alpha_{m+1}...|}{\alpha_1\alpha_2...\alpha_m\alpha_{m+1}...} \leq \frac{(\alpha_{m+1}\alpha_{m+2}...)b^{-m}}{1} \leq \frac{\frac{1}{2}bb^{-m}}{1} = \frac{1}{2}b^{1-m}$$

osservazione: $(\alpha_{m+1}\alpha_{m+2}...)$ è sicuramente minore di $b/2$.

1.3.2 Overflow e underflow

Può capitare di dover rappresentare un numero non contenuto in $\mathcal{I}$, e in questo caso ci troveremmo in una condizione di errore. I casi possibili sono:

- $|x| \geq r_2$: questa situazione è denominata *overflow*, e la sua gestione (recovery) dipende dal sistema di calcolo utilizzato. Nel nostro caso, lo standard IEEE754 prevede la definizione di una quantità, indicata con **Inf**,con eventuale segno, che rappresenta la quantità $\inf$.

- $0 \leq |x| < r_1$: questa situazione è denominata *underflow*, e la sua gestione si può tipicamente effettuare in due modi:

  1. *store 0*: si pone $fl(x) = 0$;
  2. *gradual underflow*: permette la denormalizzazione della mantissa e quindi la graduale riduzione del numero di cifre significative. In questo modo si accresce l'insieme dei numeri di macchina aggiungendogli i numeri denormalizzati, tuttavia bisogna tener presente che in questo caso l'errore ε_x aumenta oltre i limiti definiti nel teorema sopra esposto.

Possiamo adesso completare la definizione di $fl(x)$:

$$fl(x) = \begin{cases} 0,\ x = 0 \\ x(1+\varepsilon_x),\ |\varepsilon_x| \leq u,\ r_1 \leq |x| \leq r_2 \\ underflow,\ 0 \leq |x| < r_1 \\ overflow,\ |x| \geq r_2 \end{cases}$$

1.4 Lo standard IEEE 754

Lo standard IEEE 754 è stato creato per definire la rappresentazione dei numeri reali su calcolatore, allo scopo di uniformare i risultati dei calcoli eseguiti su piattaforme di calcolo differenti; cosa molto importante in vari ambiti scientifici è infatti la riproducibilità del risultato. Questo standard è stato definito e ottimizzato sulla base binaria.
Si sono scelte due rappresentazioni per i numeri reali, una a precisione singola (32 bit), e una a doppia precisione (64 bit).
Nelle due rappresentazioni i bit sono così distribuiti:

- singola precisione: esponente: 8 bit; segno: 1 bit; mantissa: 23 bit;
- doppia precisione: esponente: 11 bit; segno: 1 bit; mantissa: 53 bit;

In quanto i numeri rappresentati sono da intendersi come normalizzati, si è stabilito che α_1 non venga rappresentato, in quanto sempre uguale a 1: si ottiene un bit in più dedicabile alla mantissa, che quindi in realtà è di 24 bit per la rappresentazione a singola precisione e di 54 per quella a doppia precisione. Indicheremo con f la parte frazionaria della mantissa, ovvero l'unica parte rappresentata in macchina: $\alpha_2 ... \alpha_m$.
Per poter gestire anche i casi di eccezione, è stato deciso di dedicare il primo e l'ultimo numero rappresentabile nell'esponente a queste situazioni, ovvero:

- $e \in 1, ..., 254$: il dato è normalizzato e $\nu = 127$
- $e = 255$: rappresentazione di $\pm \inf$ se $f = 0$ (si utilizza solo il bit del segno), Not A Number (NaN) se $f \neq 0$, tipicamente il risultato di una operazione indefinita (come potrebbero essere $0 * \inf, \inf - \inf, 0/0$).
- $e = 0$: rappresentazione dello zero se $f = 0$, rappresentazione denormalizzata se $f \neq 0$: in questo caso $\nu = 126$.

per quanto riguarda la singola precisione e:

- $e \in 1, ..., 2046$: il dato è normalizzato e $\nu = 1023$
- $e = 2047$: rappresentazione di $\pm \inf$ se $f = 0$ (si utilizza solo il bit del segno), Not A Number (NaN) se $f \neq 0$, tipicamente il risultato di una operazione indefinita.
- $e = 0$: rappresentazione dello zero se $f = 0$, rappresentazione denormalizzata se $f \neq 0$: in questo caso $\nu = 1022$.

per quanto riguarda la doppia.
In questo modo si ottiene un range di numeri rappresentabili di circa $10^{-38} \div 10^{38}$ per i numeri in singola precisione e $10^{-308} \div 10^{308}$ per quelli in doppia.
La rappresentazione scelta dallo standard è per arrotondamento, ma con una piccola differenza da quello esposto precedentemente:
$fl(x)$ è definito come il numero di macchina più vicino a x, ma in caso di due numeri di macchina equidistanti, viene selezionato quello il cui ultimo bit della mantissa è 0:

$$\begin{cases} \tilde{\alpha_m} = 0 \text{ se } \alpha_{m+1} = 1 \\ \tilde{\alpha_m} = \alpha_m \text{ se } \alpha_{m+1} = 0 \end{cases}$$

Questo tipo di arrotondamento è chiamato arrotondamento al pari (round to even).

Esempi

Il più grande numero di macchina si avrà nel caso:

$$\alpha_0 = +;\ \alpha_1 = ... = \alpha_N = (b-1);\ \beta_1 = ... = \beta_s = (b-1);$$

In particolare, secondo lo standard IEEE754:
m=53, s=11, ν=1023 se il numero è normalizzato. Quindi:

$$n = ((b^0).(b^{-1})...(b^{-53}))b^{(((b^s-1)-1)-1023)} = b^{(2046-1023)}\sum_{i=0}^{52} b^{-i} = b(1-b^{-53})b^{1023} =$$

$$= 2(1-2^{-53})2^{1023} = 1.7976931e+308$$

mentre il risultato della funzione realmax di Matlab restituisce: 1.7977e+308.
Il più piccolo numero di macchina normalizzato positivo si ha invece nel caso:

$$\alpha_0 = +;\ \alpha_1 = 1, \alpha_2 = ... = \alpha_N = 0;\ \beta_1 = ... = \beta_{s-1} = 0;\ \beta_s = 1.$$

Quindi secondo lo standard:

$$n = (1.\underbrace{0...0}_{52})b^{1-1023)} = b^{-1022)} = 2.2250739e-308$$

mentre il risultato della funzione realmin di Matlab restituisce: 2.2251e-308.
Il più piccolo numero di macchina denormalizzato si ha quando:

$$n = (0.\underbrace{0...0}_{51}1)b^{0-1022} = b^{-51}b^{-1022} = b^{-1073} = 9.8813129e-324.$$

La precisione di macchina è:

$$u = \frac{1}{2}b^{1-m} = 1.110223e-16$$

1.5 Condizionamento di un problema

Operazioni elementari in aritmetica finita

É importante soffermarsi sul comportamento delle operazioni elementari in aritmetica finita. Lavorando su numeri interi, se il risultato dell'operazione cade all'interno dell'insieme di rappresentabilità, le operazioni coincidono con quelle algebriche. Considerando invece i numeri interi, le operazioni saranno definite solo su numeri di macchina e dovranno avere per risultato ancora un numero di macchina; ovvero ad esempio nel caso dell'addizione si ha che $x \oplus y = fl(fl(x) + fl(y))$. In particolare, le operazioni in macchina sui numeri reali generalmente non godono di tutte le usuali proprietà algebriche, come ad esempio la proprietà associativa e quella distributiva.

Conversione tra tipi

La conversione di tipo da intero a reale non presenta particolari problemi, in quanto, a parità di bit di rappresentazione, un intero è sempre rappresentabile in forma reale, introducendo al massimo un errore dell'ordine della precisione di macchina u: questo è infatti l'errore commesso approssimando tramite la funzione $fl(x)$. Convertire un reale in un intero invece può causare problemi, dovuti alla grande differenza nel range di rappresentabilità: circa 10^{38} per i reali e circa 10^{10} per gli interi.

Condizionamento di un problema

Supponiamo di dover risolvere un problema rappresentabile nella forma $x = f(x)$, dove x sono i dati in ingresso, y rappresenta il risultato è f è un metodo numerico. Quello che ci troveremo in realtà a risolverein macchina è un problema del tipo $\tilde{y} = \tilde{f}(\tilde{x})$; ovvero eseguiremo una funzione perturbata (con operazioni eseguite in aritmetica finita, con possibile errori di discretizzazione e/o convergenza) su un dato perturbato (errore di rappresentazione sempre presente, possibile origine sperimentale del dato). Per semplicità ci limiteremo a studiare il problema $\tilde{y} = f(\tilde{x})$, ovvero considereremo il metodo numerico come eseguito in aritmetica esatta.
Consideriamo quindi le grandezze $\tilde{x} = x(1+\varepsilon x)$ e $\tilde{y} = y(1+\varepsilon y)$ e studiamo la relazione che intercorre tra l'errore relativo in ingresso e quello in uscita, considerando un'analisi al primo ordine:

$$y + y\varepsilon_y = f(x) + f'(x)x\varepsilon_x + O(\varepsilon_x^2)$$

ovvero:

$$|\varepsilon_y| \approx \left| f'(x)\frac{x}{y} \right| |\varepsilon_x| \equiv k|\varepsilon_x|.$$

Definizione

Il valore di amplificazione k, che indica quanto gli errori iniziali possano essere amplificati sul risultato finale, è denominato *numero di condizionamento* del problema.

Significato dei valori di k:

- $k \approx 1$: gli errori sul risultato sono paragonabili a quelli sul dato iniziale. In questo caso il problema è *ben condizionato*;
- $k >> 1$: gli errori sul risultato possono essere molto più grandi di quelli sul dato iniziale. In questo caso il problema è *mal condizionato*;
- $k \approx u^{-1}$: se si utilizza una precisione di macchina u, i risultati saranno privi di significato in quanto affetti dall'errore di rappresentazione, ovvero $|\varepsilon_x| \approx u$;

1.5.1 Condizionamento delle operazioni elementari

Moltiplicazione: $y = x_1 x_2$

$$y(1+\varepsilon_y) = x_1x_2(1+\varepsilon_1)(1+\varepsilon_2) \Longrightarrow \varepsilon_y = \varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2 \Longrightarrow |\varepsilon_y| \leq |\varepsilon_1| + |\varepsilon_2| \leq 2\varepsilon_x$$

Divisione: $y = \frac{x_1}{x_2}$

$$y(1+\varepsilon_y) = \frac{x_1}{x_2}\frac{(1+\varepsilon_1)}{(1+\varepsilon_2)} \Rightarrow \varepsilon_y = \frac{\varepsilon_1 - \varepsilon_2}{1+\varepsilon_2}$$

Somma algebrica: $y = x_1 + x_2$

$$y(1+\varepsilon_y) = x_1(1+\varepsilon_1) + x_2(1+\varepsilon_2) \Longrightarrow\ y\varepsilon_y = x_1\varepsilon_1 + x_2\varepsilon_2 \Longrightarrow \varepsilon_y = |\frac{x_1\varepsilon_1 + x_2\varepsilon_2}{y}| \leq$$

$$\leq\ \frac{|x_1\varepsilon_1| + |x_2\varepsilon_2|}{|y|} \leq \frac{|x_1| + |x_2|}{|y|}\varepsilon_x = \underbrace{\frac{|x_1| + |x_2|}{|x_1 + x_2|}}_{K}\varepsilon_x$$

Mentre la moltiplicazione è ben condizionata vediamo che nel caso della somma dobbiamo differenziare due casi:

- se $x_1 \approx -x_2$ la variabile K assume valori che rendono il problema mal condizionato; nel peggiore dei casi si può incorrere nella *cancellazione numerica*.
- se x_1 e x_2 hanno segno concorde abbiamo $K = 1$.

Radice: $y = \sqrt{x}$

$$f'(x) = \frac{1}{2}x^{-\frac{1}{2}} \implies K = |\frac{\frac{1}{2}x^{\frac{1}{2}}}{x^{\frac{1}{2}}}| = \frac{1}{2}$$

Esempi

Matlab - Errori di rappresentazione

Il codice Matlab:

```
x = 0; delta = 0.1;
while x ~=1, x=x+delta, end
```

esegue un loop infinito. Questo perché il numero 0.1 non è rappresentabile in base binaria come numero finito ma solo come numero periodico, e quindi subisce un errore di approssimazione. In questo modo sommando ripetutamente 0.1 in binario non si arriva mai esattamente ad 1 e quindi il ciclo non si ferma. Questo codice potrebbe essere corretto sostituendo il controllo $x \neq 1$ con $x <= 1$, ma questo non garantisce che venga eseguito esattamente lo stesso numero di cicli che sarebbe stato eseguito dal codice precedente se avesse funzionato.

Efficienza: Individuare un algoritmo efficiente per calcolare $\sqrt{x^2 + y^2}$:

Calcolare $\sqrt{x^2 + y^2}$ eseguendo $\sqrt{xx + yy}$ non è il migliore dei modi: infatti, in caso $x = y = r_1, fl(x) = fl(y) = 0, fl(\sqrt{0}) = 0$, oppure se x è vicino ad r_2, x^2 non può essere rappresentato. Questo problema può essere risolto in modo più efficiente eseguendolo invece in questo modo:

$$m = max(|x|, |y|), \sqrt{x^2 + y^2} = m\sqrt{|\frac{x}{m}|^2 + |\frac{y}{m}|^2}.$$

che permette di effettuare il calcolo anche nelle condizioni estreme sopra descritte.

La somma algebrica non gode della proprietà associativa e distributiva:

Eseguendo in matlab il codice:

```
((eps/2+1)-1)*(2/eps)
(eps/2+(1-1))*(2/eps)
```

La variabile eps rappresenta la precisione di macchina. Più precisamente è il più largo spazio relativo tra due numeri adiacenti del sistema floating point della macchina. Questo numero è ovviamente dipendente dalla macchina, ma in macchine che supportano l'aritmetica floating point IEEE 64 bit, vale approssimativamente 2.2204e-16. Le due istruzioni sopra descritte danno come risultato:

- (eps/2+1)-1*(2/eps): ans = 0
- (eps/2+(1-1))*(2/eps): ans = 1

Ovvero non vale la proprietà associativa. Nel caso invece:

```
(1e300-1e300)*1e300
(1e300*1e300)-(1e300*1e300)
```

si hanno i seguenti risultati:

- (1e300-1e300)*1e300: ans = 0
- (1e300*1e300)-(1e300*1e300): ans = NaN

Ovvero nel primo caso l'operazione viene eseguita correttamente, mentre nel secondo si incorre in overflow e non è possibile svolgere il calcolo e quindi non vale la proprietà distributiva.

Cancellazione Numerica

Supponiamo di voler calcolare: $y = 0.12345678 - 0.12341234 \equiv 0.00004444$, utilizzando una rappresentazione decimale con arrotondamento alla quarta cifra significativa.
Otteniamo $\tilde{y} = 1.235 * 10^{-1} - 1.234 * 10^{-1} = 1 * 10^{-4}$.
L'errore relativo su y é:

$$\varepsilon_y = \frac{4.444 * 10^{-5} - 1 * 10^{-4}}{4.444 * 10^{-5}} \simeq \frac{5.560 * 10^{-5}}{4.444 * 10^{-5}} \simeq 1.25$$

mentre essendo $\varepsilon_{x_1} = 3.5 * 10^{-4}$ e $\varepsilon_{x_2} = -1 * 10^{-4}$, si ha che

$$k = \frac{|3.5 * 10^{-4}| + |-1 * 10^{-4}|}{|3.5 * 10^{-4} - 1 * 10^{-4}|} \simeq 5.5 * 10^{3}$$

Il problema risulta essere mal condizionato.

Capitolo 2

Radici di una equazione

In questo capitolo tratteremo la risoluzione dell'equazione:

$$f(x) = 0, \quad x \in \mathbf{r}$$

in cui $f : \mathbb{R} \to \mathbb{R}$ è una funzione assegnata; ovvero siamo interressati a trovare, se esiste, uno zero reale della funzione. In generale ci possiamo trovare in tre diverse situazioni:

- $f(x)$ ha un numero finito di soluzioni; ad esempio il caso $f(x) = (x-1)(x-2)$.
- $f(x)$ non ha soluzioni reali; ad esempio $f(x) = x^2 + 1$.
- $f(x)$ ha un numero infinito di soluzioni; ad esempio $f(x) = sin(x)$.

In particolare ci limiteremo a lavorare su funzioni tali che, definito $[a, b]$ intervallo in $\mathbb{R}$ con $a < b$, queste godano di certe proprietà:

- $f(x)$ è continua nell'intervallo $[a, b]$
- $f(a)f(b) < 0$.

Questo garantisce l'esistenza di una radice di f all'interno dell'intervallo $[a, b]$, ovvero la funzione, passando da positiva a negativa all'interno di questo intervallo, deve necessariamente attraversare l'asse delle ascisse in almeno un punto.

2.1 Il metodo di bisezione

Come primo passo dobbiamo scegliere un punto interno all'intervallo che sia una miglior approssimazione della radice ($\bar{x}$), in particolare scegliamo il punto medio: $x_1 = \frac{a+b}{2}$.
A questo punto possono verificarsi solo tre casi:

- $f(x_1) = 0$: abbiamo trovato la radice e quindi risolto il problema;
- $f(a)f(x_1) < 0$: la funzione interseca l'asse delle ascisse prima del punto x_1 e si può ripetere il procedimento sull'intervallo $[a, x_1]$
- $f(x_1)f(b) < 0$: la funzione interseca l'asse delle ascisse dopo il punto x_1 e si può ripetere il procedimento sull'intervallo $[x_1, b]$.

2.1.1 Criteri di arresto:

Con questo metodo si ha che $|\bar{x} - x_1| \leq \frac{b-a}{2}$. Ragionando per induzione, indicando con $[a_i, b_i]$ l'intervallo di confidenza e con $x_i = (a_i + b_i)/2$ l'approssimazione della radice all'intervallo i-esimo, si ha che:

$$|\bar{x} - x_i| \leq \frac{b-a}{2^i}$$

Ovvero ad ogni iterazione si dimezza la distanza massima della serie dalla soluzione. In questo modo, si può pensare di voler ottenere il calcolo di una approssimazione della radice che disti al più $tolx$ dalla radice,e si ha che questa si raggiunge in un numero di iterazioni:

$$imax \equiv [log_2(b-a) - log_2(tolx)]$$

Adesso sappiamo che sicuramente, dopo $imax$ itarazioni, abbiamo raggiunto la precisione richiesta; ma possiamo ancora stabilire un criterio efficace per accorgersi se si è arrivati sufficientemene vicini alla soluzione prima di $imax$ iterazioni. In linea di massima, se $f(\bar{x}) = 0$, per la continuità di f si deve avere che il valore che la funzione assume in prossimità di x debba assumere valori vicini a 0. In questo modo possiamo pensare ad un controllo di arresto del tipo $|f(x)| < tolf$ dove $tolf$ deve essere scelta in modo da avere $|x - \bar{x}| < tolx$. Supponendo $f \in C^{(1)}$, sviluppando in serie di Taylor:

$$f(x) \approx f(\bar{x}) + f'(\bar{x})(x - \bar{x}) = f'(\bar{x})(x - \bar{x}) \quad ovvero \quad |x - \bar{x}| \approx \frac{f(x)}{|f'(\bar{x})|}$$

e quindi $tolf \approx |f'(\bar{x})|tolx$.

Osservazione:

Tanto più grande sarà $|f'(\bar{x})|$, tanto meno stringente dovrà essere il criterio di arresto sulla funzione. Infatti si ha che:

$$|x_i - \bar{x}| \approx \frac{f(x_i) - f(\bar{x})}{|f'(\bar{x})|} \quad ovvero \quad k = \frac{1}{|f'(\bar{x})|}.$$

dove k rappresenta il numero di condizionamento del problema. Questo metodo sarà ben condizionato in presenza di funzioni che hanno una derivata prima molto grande nei pressi della radice, ma mal condizionato quando questa si avvicina o tende a zero.

Implementazione

Considerando che una buona approssimazione di $f'(\bar{x})$ è ricavabile, a costo molto basso, in questo modo:

$$f'(\bar{x}) \approx \frac{f(b_i) - f(a_i)}{b_i - a_i}$$

si può pensare al seguente algoritmo per l'applicazione del metodo di bisezione:

```
%Questa function determina uno zero della funzione in ingresso utilizzando il
%metodo di bisezione.
%
%  [x,i]=bisezione(f,a,b,tolx)
%
%Questo metodo ha come input:
%  f: la funzione di cui si vuol trovare uno zero
%  tolx: tolleranza assoluta sul valore dello zero
% a: estremo sinistro dell'intervallo
% b: estremo destro dell'intervallo
%
%E restituisce:
%  x: zero della funzione
%  i: numero di iterazioni fatte
% imax: il numero massimo stimato di iterazioni



function [x,i,imax]=bisezione(f,a,b,tolx)
fa = feval(f,a);
fb = feval(f,b);

% controllo se ha senso proseguire
if fa == 0
    x = a;
    return
end
if fb == 0
    x = b;
    return
end

if abs(fa) == Inf | abs(fb) == Inf
    error('La funzione non e valutabile in uno degli estremi.')
    return
end
if fa*fb > 0
    error('La funzione non soddisfa la condizione f(a)*f(b) < 0.')
    return
end

% inizializzazione del ciclo e calcolo del limite massimo di iterazioni
x=(a+b)/2;
fx = feval(f,x);
imax = ceil(log2(b-a)-log2(tolx));

for i = 2:imax
    f1x = ((fb-fa)/(b-a));
    if abs(fx)<=tolx*abs(f1x);
       break
    elseif fa*fx<0
        b=x;
        fb=fx;
    else
        a=x;
        fa=fx;
    end
x=(a+b)/2;
fx=feval(f,x);
end
```

Costo computazionale

Per quanto riguarda il costo di questo algoritmo in termini di operazioni floating-point, si vede facilmente dal codice che questo algoritmo ad ogni ciclo esegue esattamente 3 somme, 4 moltiplicazioni ed una valutazione di funzione, tutto questo al massimo imax volte. Tenendo conto che imax è definito come $log_2(b-a)-log_2(tolx)$, si può affermare che questo algoritmo ha un costo di andamento grossomodo logaritmico all'allargarsi dell'intervallo di confidenza. Il costo in termini di memoria è anch'esso molto contenuto, si utilizzano infatto soltanto poche variabili.

2.1.2 Ordine di convergenza

Definito come $e_i = x_i - \bar{x}$ l'errore commesso al passo i-esimo nell'approssimazione fornita da un dato metodo numerico, questo medoto è convergente se:

$$\lim_{i\to\infty} e_i = 0$$

Il metodo ha ordine di convergenza $p \in \mathbb{R}$ se p è il più grande numero positivo per cui:

$$\lim_{i\to\infty} \frac{|e_{i+1}|}{|e_i|^p} = c < \infty$$

dove c e detta costante asintotica dell'errore. In particolare si avranno metodi che convergono linearmente quando $p = 1$, esponenzialmente (quadraticamente, cubicamente...) per $p > 1$, e metodi non convergenti se $p < 1$.

$$|e_{i+1}| \approx c|e_i|, \quad |e_{i+2}| \approx c|e_{i+1}| \approx c^2|e_i|$$

ovvero, per induzione:

$$|e_{i+k}| \approx c^k|e_i|$$

Abbiamo quindi che per i molto grandi, ovvero dopo molte iterazioni, anche k è molto grande, e si deve avere che $0 \leq c \leq 1$ altrimenti la funzione sarebbe divergente.

2.2 Il metodo di Newton

Il metodo di Newton si basa sull'idea di sfruttare, come successiva approssimazione della radice, il punto di intersezione della retta tangente alla funzione nel punto $(x_0, f(x_0))$ con l'asse delle ascisse, considerando x_0 l'approssimazione iniziale. Consideriamo quindi $y = f(x_i) + f'(x_i)(x - x_i)$, ponendo y a zero si ha che:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

che è definita per $f'(x_0) \neq 0$. Possiamo estendere il ragionamento a tutti i passi successivi:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \quad i = 0, 1, 2...$$

Teorema

Se x_i è la successione sopre descritta, che converge a $\bar{x}$, zero semplice di $f(x)$ funzione di classe $\mathcal{C}^2$, allora il metodo di Newton converge almeno quadraticamente.
Dim.
Per ipotesi abbiamo che $x_i \to \bar{x}$ per $i \to \infty$. Quindi:

$$0 = f(\bar{x}) = f(x_i) + f'(x_i)(\bar{x} - x_i) + \frac{1}{2}f''(\psi_i)(\bar{x} - x_i)^2 =$$

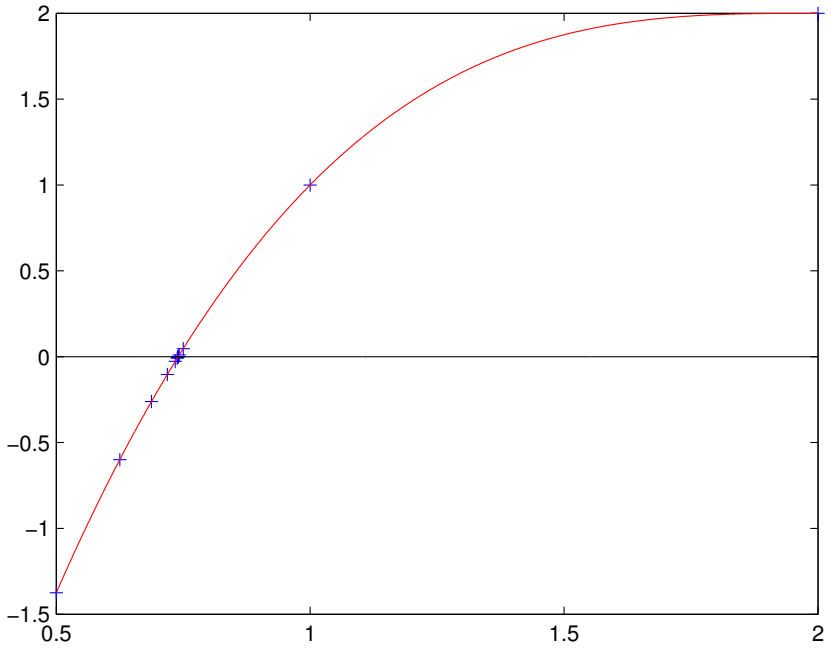

Figura 2.1: Esempio di funzionamento del metodo di bisezione che mostra in che punti viene valutata la funzione $(x-2)^3+2$ con l'intervallo di partenza $[0,4]$; gli estremi 0 e 4 sono stati esclusi dal grafico per migliorare la visualizzazione.

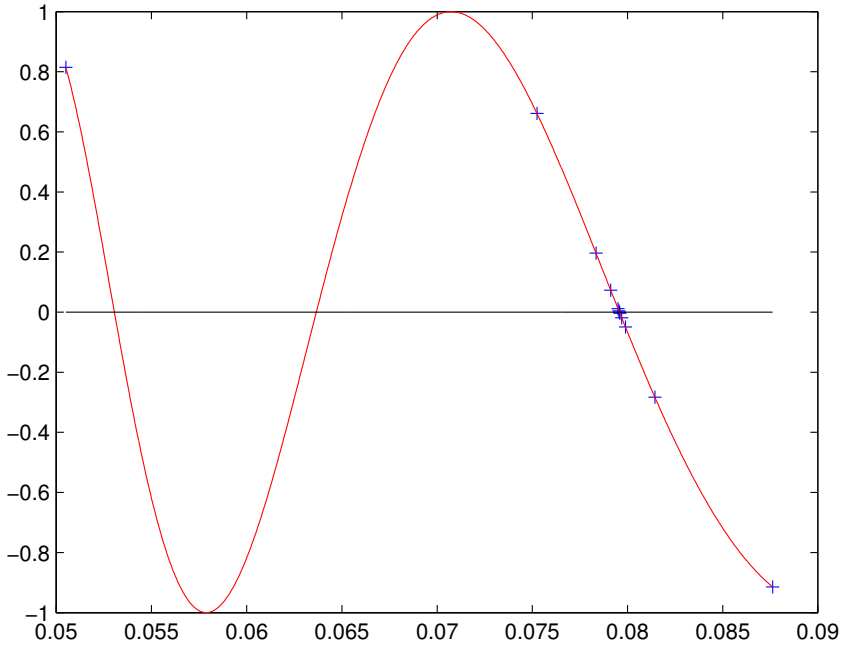

Figura 2.2: Esempio di funzionamento del metodo di bisezione che mostra come, nel caso la funzione abbia più di uno zero all'interno dell'intervallo di confidenza, questo converga verso uno di essi. In questo caso la funzione è $y = sin(\frac{1}{x})$ con l'intervallo di partenza $[10^{-3}, 10^{-1}]$.

$$= f'(x_i)\left(\frac{f(x_i)}{f'(x_i)} + (\bar{x} - x_i)\right) + \frac{1}{2}f''(\psi_i)(\bar{x} - x_i)^2 = f'(x_i)(\underbrace{-x_{i+1} + \bar{x}}_{-e_{i+1}}) + \frac{1}{2}f''(\psi_i)e_i^2$$

$$\frac{e_{i+1}}{e_i^2} = \frac{1}{2}\frac{f''(\psi_i)}{f'(x_i)}$$

che è ben definita in quanto $f'(x_i)$ è sicuramente diverso da zero perché stiamo trattando il caso di una radice semplice e siamo in un opportuno intorno di $\bar{x}$, e se il metodo converge si ha:

$$\lim_{i\to\infty} \frac{1}{2}\frac{f''(\psi_i)}{f'(x_i)} = \frac{1}{2}\frac{f''(\bar{x})}{f'(\bar{x})}$$

Osservazione:

Se $x_i \to \bar{x}$ con $i \to \infty$ radice di molteplicità $m > 1$, allora l'ordine di convergenza del metodo di Newton è solo lineare ($p = 1$) con $c = (m-1)/m$

2.2.1 Convergenza locale

Rimane ancora da stabilire quali siano le condizione sufficienti a garantire la convergenza del metodo verso la soluzione. Per prima cosa è necessario formalizzare un metodo iterativo per approssimare una radice $\bar{x}$ dell'equazione $f(x) = 0$:

$$x_{i+1} = \Phi(x_i), \quad i = 0, 1, 2, ...,$$

ed ovviamente la radice deve essere un punto fisso per questa funzione di iterazione. Possiamo quindi definire la funzione di iterazione per il metodo di Newton:

$$\Phi(x) = x - \frac{f(x)}{f'(x)}.$$

Teorema del punto fisso

Sia $\Phi(x)$ la funzione di iterazione che definisce un metodo numerico. Supponiamo che $\exists \quad \delta > 0, 0 \leq L < 1$ tali che $\forall x, y \in (\bar{x} - \delta, \bar{x} + \delta) \equiv \mathcal{I}; \quad |\Phi(x) - \Phi(y)| \leq L|x - y|$ allora:

1. $\bar{x}$ è l'unico punto fisso di Φ in $\mathcal{I}$
2. se $x_0 \in \mathcal{I} \Rightarrow x_i \in \mathcal{I} \quad \forall i \geq 0$
3. $x_i \to \bar{x}$ per $i \to \infty$

Dim(1).
Supponiamo per assurdo che $\exists \quad \tilde{x} \quad | \quad \tilde{x} \equiv \Phi(\tilde{x}) \in \mathcal{I}$ ulteriore punto fisso per Φ, ovviamente $\bar{x} \neq \tilde{x}$. Ma allora si ha che: $0 < |\bar{x} - \tilde{x}| = |\Phi(\bar{x}) - \Phi(\tilde{x})| \leq L|\bar{x} - \tilde{x}|$ ma essendo $L < 1$ si ha che $|\bar{x} - \tilde{x}| < |\bar{x} - \tilde{x}|$, assurdo.
Dim(2).
Sia $x_0 \in \mathcal{I} \Longleftrightarrow |\bar{x} - x_0| < \delta$.
Allora: $|\bar{x} - x_i| < \delta$ e $|\bar{x} - x_i| = |\Phi(\bar{x}) - \Phi(x_0)| \leq L|\bar{x} - x_0| < L\delta < \delta$
e $x_i \in \mathcal{I}$.
Dim(3).
$|x_i - \bar{x}| = |\Phi(x_{i-1}) - \Phi(\bar{x})| \leq L|x_{i-1} - \bar{x}|$, ovvero:
$|x_i - \bar{x}| = L|\Phi(x_{i-2}) - \Phi(\bar{x})| \leq L^2|x_{i-2} - \bar{x}|$
Ragionando per induzione possiamo quindi concludere che:
$|x_i - \bar{x}| \leq L^i|x_0 - \bar{x}|$, che tende a zero per i che tende a infinito.

Se si assume $f(x) \in C^{(2)}$, tramite questo teorema è possibile dimostrare la convergenza locale del metodo di Newton. In questo caso infatti la funzione di iterazione è di classe $C^{(1)}$ e $\Phi'(\bar{x}) = 0$. Fissato quindi un arbitrario $L \in [0, 1)$, esisterà un $\delta > 0$ tale che $|\Phi'(x)| \leq L, \forall x \in \mathcal{I} \equiv (\bar{x} - \delta, \bar{x} + \delta)$. Sviluppando in serie di Taylor al primo ordine si ottiene che:

$$\forall x, y \in \mathcal{I}: \quad |\Phi(x) - \Phi y| = |\Phi(x) - \Phi(x) - \Phi'(\varepsilon)(y - x)| \leq L|x - y|,$$

con ε compreso tra x e y ovvero apppartenente ad $\mathcal{I}$. La funzione di iterazione soddisfa le ipotesi del teorema, e quindi il metodo di Newton è localmente convergente se $x_0 \in \mathcal{I}$

2.2.2 Criteri di arresto

Differentemente dal metodo di bisezione, a causa della convergenza solo locale del metodo di Newton, non è possibile stabilire a priori il numero massimo di iterazioni necessarie per raggiungere la soluzione con l'accuratezza richiesta. Come nel caso precedente vorremmo stabilire un criterio sulla dimensione dell'errore, ovvero vogliamo fermarci quando $|e_i| \leq tolx$ dove $tolx$ rappresenta appunto la tolleranza che siamo disposti ad accettare intorno alla soluzione. Notiamo che nel caso di convergenza verso radici semplici il metodo di Newton converge quadraticamente ed in prossimità della radice si ha:

$$|x_{i+1} - x_i| = |\underbrace{x_{i+1} - \bar{x}}_{-e_{i+1}} + \underbrace{\bar{x} - x_i}_{e_i}| = |e_i - e_{i+1}| \approx |e_i| \leq tolx$$

e quindi un controllo del tipo $|x_{i+1} - x_i| < tolx$ è appropriato ed equivalente al controllo $|f(x_i)| \leq |f'(x_i)|tolx$ in quanto $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$. Bisogna notare però che questo controllo essenzialmente non è altro che una verifica sull'errore assoluto su x, ovvero $tolx \simeq O(x)$. Benché questo possa risultare appropriato per valori piccoli di x, non lo è necessariamente anche per valori grandi, dove risulta molto più efficiente un controllo sull'errore relativo. Possiamo quindi definire la grandezza "errore relativo su x" come $|e_i| \leq rtol|\bar{x}| \approx rtol|x_{i+1}|$, ed un criterio di arresto del tipo:

$$\frac{|x_{i+1} - x_i|}{\underbrace{tolx}_{\text{tolleranza assoluta}} + \underbrace{rtolx|x_{i+1}|}_{\text{tolleranza relativa}}} \leq 1$$

che è un criterio di arrsto ibrido che si arresta quando è soddisfatto il criterio più stringente. In particolare si ha che per valori piccoli di x, $rtolx|x_{i+1}|$ è un valore piccolo rispetto ad x e quindi il criterio assoluto predomina, viceversa invece per valori di x grandi; ponendo $rtolx$ uguale a zero questo criterio ritorna equilvalente a $|x_{i+1} - x_i| < tolx$.

Convergenza lineare

Quando la convergenza del metodo è solo lineare, il criterio di arresto può essere modificato. Si ha infatti che:

$$|x_{i+1} - x_i| = |e_i - e_{i+1}| \approx |e_i|(1 - c) \text{ con } c = \frac{m - 1}{m}$$

Noi vogliamo $|e_i| < tolx$ e quindi:

$$|e_i| = \frac{|x_{i+1} - x_i|}{(1 - c)} \leq tolx$$

Ovvero abbiamo ottenuto un criterio tanto più efficiente quanto più la molteplicità della radice è alta ($c \approx 1$). Noi conosciamo x_{i+1}, e vogliamo che $|e_{i+1}| \leq tolx$ ma sappiamo che $|e_{i+1}| = c|e_i|$ e possiamo quindi controllare $\frac{c}{1-c}|x_{i+1} - x_i| \leq tolx$. Il problema è che non conosciamo c. Possiamo però calcolare le prime tre iterazioni (x_0, x_1, x_2) in questo modo: $|x_1 - x_0| \approx (1-c)|e_0|$; $|x_2 - x_1| \approx (1-c)|e_1| \approx (1-c)c|e_0|$; e quindi $c \approx \dfrac{|x_2 - x_1|}{|x_1 - x_0|}$.

Implementazione

Visto quanto detto fino ad ora, possiamo pensare alla seguente implementazione del metodo di Newton:

```
%Questo metodo determina uno zero della funzione in ingresso utilizzando il
%metodo di Newton.
%
%  [x,i]=newton(f,f1,x0,itmax,tolx,rtolx)
%
%Questo metodo prende in input:
%  f: la funzione di cui si vuol trovare uno zero
%  f1: la derivata della funzione
%  imax: numero massimo di iterazioni consentite
%  tolx: tolleranza assoluta sul valore dello zero
% rtolx: tolleranza relativa

%
%Questo metodo restituisce:
%  x: zero della funzione
%  i: numero di iterazioni fatte

function [x,i]=newton(f,f1,x0,itmax,tolx,rtolx)

i=0;
fx = feval(f,x0);

% se siamo particolarmente fortunati...
if fx==0
    x=x0;
    return
end

f1x = feval(f1,x0);
if f1x==0
      error('La derivata prima ha assunto valore zero, impossibile continuare!')
end
x= x0-fx/f1x;

while (i<itmax) & ((abs(x-x0)/(tolx+rtolx*(abs(x))))>1)
    i = i+1;
    x0 = x;
    fx = feval(f,x0);
    f1x = feval(f1,x0);
    %Se la derivata vale zero non possiamo continuare, rimane solo da
    % controllare che non si abbia raggiunto ugualmente una soluzione
    % nelle condizioni di tolleranza richieste.
    if f1x==0
        if fx == 0
             return
         elseif ((abs(x-x0)/(tolx+rtolx*(abs(x))))<=1)
            return
        end
      error('La derivata prima ha assunto valore zero, impossibile continuare!')
```

```
    end
    x = x0-fx/f1x;
end
%Se il ciclo sopra e' terminato perche' abbiamo raggiunto il limite massimo di
%iterazioni, non abbiamo ottenuto un risultato accettabile.
if ((abs(x-x0)/(tolx+rtolx*(abs(x))))>1), disp('Il metodo non converge.'), end
```

Costo Computazionale

Rispetto al metodo di bisezione il metodo di Newton ha un costo leggermente più alto: questo richiede infatti 7 operazioni floating-point e due valutazioni di funzione ad ogni iterazione. In questo caso il costo massimo della determinazione dello zero non è determinabile a priori, e dipende dalla scelta di itmax effettuata dall'utente. Il costo in termini di memoria, come nel caso del metodo di bisezione è anch'esso molto contenuto e si utilizzano soltanto poche variabili.

2.2.3 Ripristino della convergenza quadratica in caso di radici multiple

Molteplicità di una radice:

Una radice $\bar{x}$ ha moltiplicità esatta m se:

$$f(\bar{x}) = f'(\bar{x}) = ... = f^{m-1}(\bar{x}) = 0, \quad , f^{(m)}(\bar{x}) \neq 0.$$

Osservazione:

Se $f(x)$ è sviluppabile in serie di Taylor in $\bar{x}$, sua radice di molteplicità esatta m, allora è scrivibile nella forma $f(x) = (x - \bar{x})^m g(x)$ con $g(x)$ ancora sviluppabile in serie di Taylor in $\bar{x}$ e tale che $g(\bar{x}) \neq 0$.

Molteplicità nota

Supponiamo di avere una radice $\bar{x}$ di molteplicità nota m. Allora $f(x)$ sarà del tipo $(x - \bar{x})^m$. Avendo x_0 approssimazione di $\bar{x}$:

$$x_0 \to x_1 = x_0 - \frac{(x_0 - \bar{x})^m}{m(x_0 - \bar{x})^{(m-1)}} = x_0 - \frac{1}{m}(x_0 - \bar{x}) = x_0 - m\frac{f(x_0)}{f'(x_0)}$$

ovvero

$$x_{i+1} = x_i - m\frac{f(x_i)}{f'(x_i)}, \quad i = 0, 1, 2...$$

In questo caso particolare si riesce ad ottenere la convergenza alla radice in un solo passo iterativo, utilizzando il *fattore di correzione* m.
In generale, $f(x)$ è una funzione del tipo $(x - \bar{x})^m g(x)$ (con $g(x)$ qualsiasi) e non si riesce quindi ad ottenere il risultato in un solo passo iterativo come nel caso particolare detto sopra; è dimostrabile che questa modifica riesce però a ripristinare la convergenza quadratica del metodo di Newton verso radici di molteplicità nota.

Implementazione

```
%Questo metodo determina uno zero della funzione in ingresso utilizzando il
%metodo di Newton nel caso di una radice di molteplicita' nota.
%
```

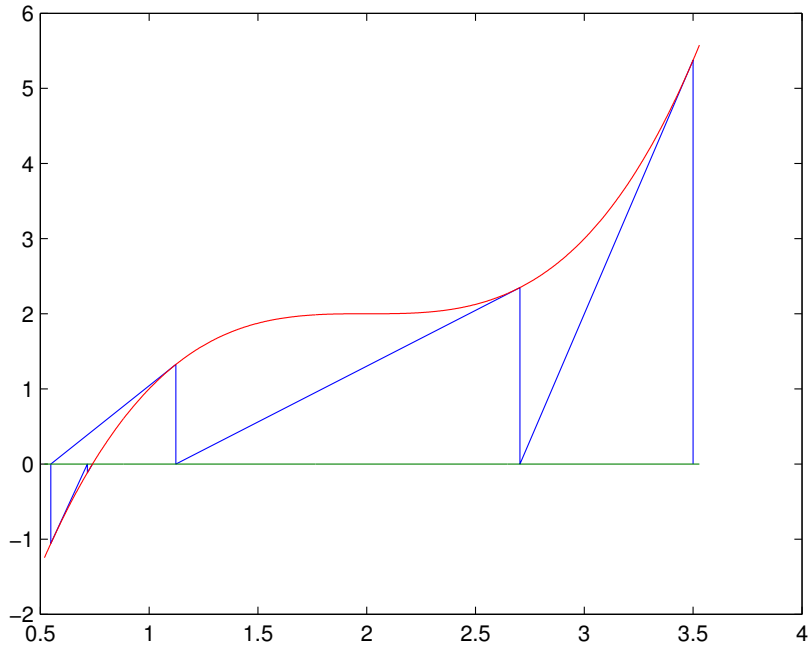

Figura 2.3: Esempio di funzionamento del metodo di Newton che ne mostra il comportamento nella determinazione dello zero della funzione $(x-2)^3+2$ con punto di innesco $x_0 = 3.5$.

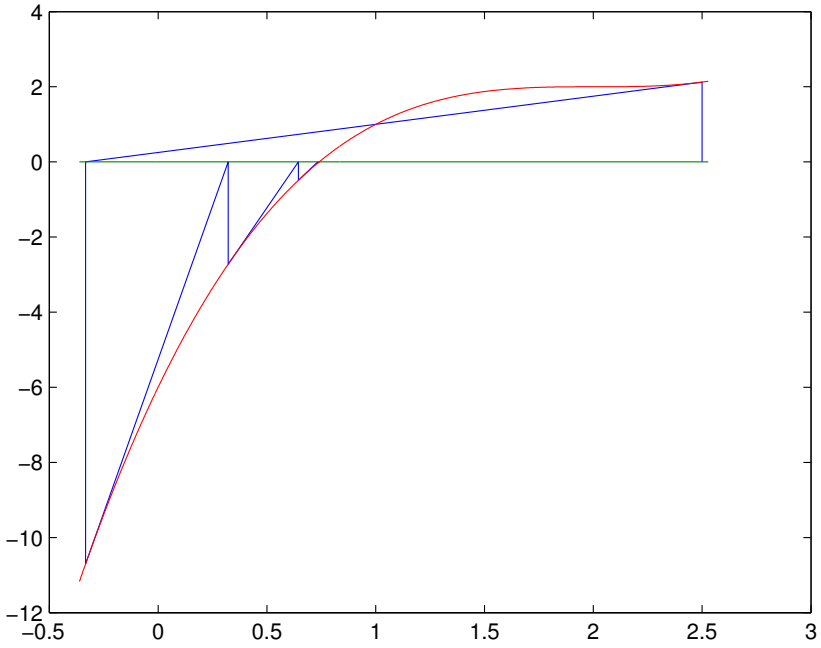

Figura 2.4: Ecco come cambia il comportamento del metodo di Newton nella determinazione dello zero della funzione $(x-2)^3+2$ se il punto di innesco è invece $x_0 = 2.5$.

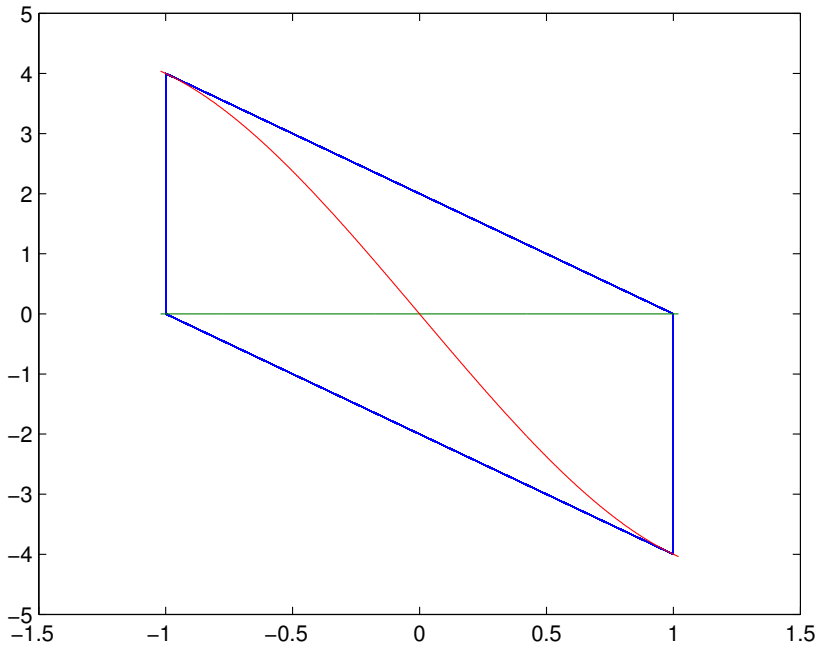

Figura 2.5: In questo caso il metodo di Newton non converge, stiamo cercando la radice di $x^3 + 5x$ a partire dal punto $x_0 = 1$ ma il metodo genera una successione infinita del tipo $\{1, -1, 1, -1, ..\}$,

```
%  [x,i]=newtonmoltnota(f,f1,x0,itmax,tolx,rtolx,m)
%
%Questo metodo prende in input:
%  f: la funzione di cui si vuol trovare uno zero
%  f1: la derivata della funzione
%  imax: numero massimo di iterazioni consentite
%  tolx: tolleranza assoluta sul valore dello zero
%  rtolx: tolleranza relativa sul valore dello zero
%       m: la molteplicita' della radice
%
%Questo metodo restituisce:
%  x: zero della funzione
%  i: numero di iterazioni fatte

function [x,i]=newtonmoltnota(f,f1,x0,itmax,tolx,rtolx,m)
i=0;
fx = feval(f,x0);
if fx==0
    x=x0;
    return
end
f1x = feval(f1,x0);
if f1x==0
    error('La derivata prima ha assunto valore zero, impossibile continuare!')
    return;
end
x= x0-(m*fx)/f1x;
while (i<itmax) & ((abs(x-x0)/(tolx+rtolx*(abs(x))))>1)
    i = i+1;
    x0 = x;
    fx = feval(f,x0);
    f1x = feval(f1,x0);
    if f1x==0
        if fx == 0
             return
         elseif ((abs(x-x0)/(tolx+rtolx*(abs(x))))<=1)
            return
        end
      error('La derivata prima ha assunto valore zero, impossibile continuare!')
    end
    x = x0-(m*fx)/f1x;
end
%Se il ciclo sopra e' terminato perche' abbiamo raggiunto il limite massimo di
%iterazioni, non abbiamo ottenuto un risultato accettabile.
if ((abs(x-x0)/(tolx+rtolx*(abs(x))))>1), disp('Il metodo non converge.'), end
```

Molteplicità non nota

Supponiamo invece di avere una radice $\bar{x}$ di molteplicità ignota. In questo caso al generico passo i-esimo si ha che $e_i \approx ce_{i-1}$ ed $e_{i+1} \approx ce_i$, con c costante asintotica dell'errore, incognita. Dalla combinazione delle due si ottiene che $e_{i+1}e_{i-1} \approx e_i^2$ e quindi $(x_{i+1} - \bar{x})(x_{i-1} - \bar{x}) \approx (x_i - \bar{x})^2$. Se si suppone esatta l'uguaglianza, si ha che:

$$\bar{x} \approx \frac{x_{i+1}x_{i-1} - x_i^2}{x_{i+1} - 2x_i + x_{i-1}} \equiv \bar{x}^{(i)}$$

In questo modo si può pensare ad una macchinetta iterativa che agisce su due livelli:

1. Livello interno in cui vengono eseguiti due passi del metodo di Newton, ovvero a partire da una approssimazione iniziale $x_0^{(0)}$ si calcolano $x_1^{(0)}, x_2^{(0)}$ approssimazioni di Newton alla prima iterazione,

2. Livello esterno in cui, con la formula:

$$x_0^{(1)} = \frac{x_2^{(0)} x_0^{(0)} - (x_1^{(0)})^2}{x_2^{(0)} - 2x_1^{(0)} + x_0^{(0)}}$$

   si ottiene una successiva approssimazione più accurata $x_0^{(1)}$ della radice da utilizzare per il successivo ciclo interno. Questo sistema, chiamato *"metodo di accelerazione di Aitken"*, permette di ripristinare la convergenza quadratica del metodo di newton nel caso di radici multiple di molteplicità ignota, al prezzo di aumentare il numero di operazioni da eseguire ad ogni iterazione (ciclo a due livelli).

Implementazione

```
%Metodo di accelerazione di Aitken
%
% [x,i]=aitken(f,f1,x0,itmax,tolx,rtolx)
%
%Questo metodo prende in input:
%  f: la funzione di cui si vuol trovare uno zero
%  f1: la derivata della funzione
% x0: ascissa di partenza
%  imax: numero massimo di iterazioni consentite
%  tolx: tolleranza assoluta sul valore dello zero
% rtolx: tolleranza relativa
%
%Questo metodo restituisce:
%  x: zero della funzione
%  i: numero di iterazioni fatte

function [x,i]=aitken(f,f1,x0,itmax,tolx,rtolx)

i=0;
fx = feval(f,x0);

% se siamo particolarmente fortunati...
if fx==0
    x=x0;
    return
end

f1x = feval(f1,x0);
if f1x==0
   error('La derivata prima ha assunto valore zero, impossibile continuare!')
end
x= x0-fx/f1x;

go = 1;
while (i<itmax) & go
    i = i+1;
```

```
    x0 = x;
    fx = feval(f,x0);
    f1x = feval(f1,x0);
    if f1x==0
    %In questo caso non possiamo andare avanti, rimane solo da controllare
    %se per caso abbiamo trovato una soluzione esatta o almeno nella tolleranza
    %richiesta
        if fx == 0
     %Abbimo trovato una soluzione esatta
     return
     elseif ((abs(x-x0)/(tolx+rtolx*(abs(x))))<=1)
     %Abbiamo trovato una soluzione nella tolleranza richiesta
             return
end
%Evitiamo una divisione per zero.
      error('La derivata prima ha assunto valore zero, impossibile continuare!')
    end
    x1 = x0-fx/f1x;
    fx = feval(f,x1);
    f1x = feval(f1,x1);
    if f1x==0
        if fx == 0
            return
    elseif ((abs(x-x0)/(tolx+rtolx*(abs(x))))<=1)
return
        end
      error('La derivata prima ha assunto valore zero, impossibile continuare!')
    end
    x = x1 - fx/f1x;
    t = ((x-2*x1)+x0);
    %Questo controllo serve ad aggirare il problema della cancellazione numerica
    %in quanto la sottrazione appena fatta è mal condizionata (stiamo sottraendo
    %due valori molto vicini). In caso t sia 0 non possiamo proseguire, ma
    %vale la pena controllare di aver trovato una soluzione. Il controllo
    %sulla tolleranza delle ascisse non ha senso in quanto sarebbe anch'esso
    %soggetto alla cancellazione numerica e quindi inaffidabile.
    if t == 0
        if feval(f,x) == 0
            return
        end
        error('Impossibile determinare la radice nella tolleranza desiderata')
    end
        x = (x*x0-x1^2)/t;
go = ((abs(x-x0)/(tolx+rtolx*(abs(x))))>1);
end
%Se il ciclo sopra e' terminato perche' abbiamo raggiunto il limite massimo di
%iterazioni, non abbiamo ottenuto un risultato accettabile.
if go, disp('Il metodo non converge.'), end
```

Costo computazionale

Come descritto nella teoria, questo algoritmo effettua, ad ogni iterazione, due passi del metodo di Newton ed un passo di accelerazione, per un costo totale quindi di 16

operazioni floating point e 4 valutazioni di funzione. Per quanto riguarda l'occupazione di memoria, anche qui usiamo solo poche variabili.

2.3 Metodi quasi-Newton

I metodi detti quasi-Newtoniani sono delle varianti del metodo di Newton che permettono di risparmiare il calcodo della derivata prima della funzione ad ogni passo iterativo. Questi metodi sono descrivibili tramite uno schema generale che li accomuna:

$$x_{i+1} = x_i - \frac{f(x_i)}{\varphi_i}, \quad i = 1, 2, ..., \quad \varphi_i \approx f'(x)$$

2.3.1 Metodo delle corde

Questo metodo si basa sull'osservazione che, in prossimità della radice, la derivata prima della funzione in esame varia di poco, e si può quindi pensare di calcolarla una sola volta e di utilizzare poi un fascio di rette parallele a questa per approssimarla nelle iterazioni successive. Supponendo di essere in presenza su una funzione sufficientemente regolare ed abbastanza vicini alla radice, si può definire il metodo come:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_0)}, \quad i = 0, 1, 2, ...,$$

Il vantaggio di questo metodo risiede nel basso costo computazionale per iterazione (si valuta solo una volta $f'(x)$ e una sola volta per iterazione $f(x)$), ma si può dimostrare che l'ordine di convergenza è solo lineare.

Implementazione

```
%Questo metodo determina uno zero della funzione in ingresso utilizzando il
%metodo delle corde.
%
%  [x,i]=corde(f,f1,x0,itmax,tolx,rtolx)
%
%Questo metodo prende in input:
%  f: la funzione di cui si vuol trovare uno zero
%  f1: la derivata della funzione
%  itmax: numero massimo di iterazioni consentite
%  tolx: tolleranza assoluta sul valore dello zero
%  rtolx: tolleranza relativa sul valore dello zero
%
%Questo metodo restituisce:
%  x: zero della funzione
%  i: numero di iterazioni fatte



function [x,i] = corde(f,f1,x,itmax,tolx,rtolx)

i=0;
f1x = feval(f1,x);
if f1x==0
error('La derivata prima ha valore nullo, impossibile continuare')
end
```

```
go = 1;
while (i<itmax) & go
    i = i+1;
    x0 = x;
    x = x0 - (feval(f,x0)/f1x);
    go = ((abs(x-x0)/(tolx+rtolx*(abs(x))))>1);
end
```

Costo computazionale

Questo metodo ad ogni ciclo iterativo svolge soltanto 7 operazioni floating point ed una valutazione di funzione, il costo per iterata è quindi molto basso. Bisogna inoltre notare come la convergenza del metodo sia molto legata al tipo di funzione e alla scelta del punto di partenza; se questo non è abbastanza vicino alla radice oppure la funzione non è abbastanza regolare, il mtodo può convergere molto lentamente.

2.3.2 Metodo delle secanti

Tenendo conto della definizione della derivata prima mediante rapporto incrementale, si può pensare di sfruttare due approssimazioni successive (ovvero due precedenti valutazioni di $f(x)$) per approssimarla. Considerando quindi di essere all'iterazione i-esima e di aver già calcolato $x_{i-1}, x_i, f(x_{i-1}), f(x_i)$, si approssima $f'(x)$ in questo modo:

$$\varphi(x) \approx f'(x) = \frac{f(x+\delta) - f(x)}{\delta} = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

Il metodo iterativo risultante:

$$x_{i+1} = \frac{f(x_i)x_{i-1} - f(x_{i-1})x_i}{f(x_i) - f(x_{i-1})}, \quad i = 1, 2, ...,$$

con x_0, x_1 approssimazioni iniziali assegnate.
In questo modo si ha la possibilità di calcolare una sola volta per itearazione il valore di $f(x)$, ottenendo così un metodo con costo per iterazione molto basso; in più, quando la funzione stà convergendo si ha che x_n è molto vicino a x_{n-1}, e il rapporto incrementale risulta una buona approssimazione per la derivata. Questo metodo è caratterizzato da un'ordine di convergenza $1 \leq p \leq 2$; in particolare si può dimostrare che $p = \frac{\sqrt{5}+1}{2} \approx 1,618$ nel caso di radici semplici, ma è solo lineare nel caso di radici multiple.

Implementazione

```
%Questo metodo determina uno zero della funzione in ingresso utilizzando il
%metodo delle secanti.
%
% [x,i] = secanti(f,f1,x0,itmax,tolx,rtolx)
%
%Questo metodo prende in input:
%  f: la funzione di cui si vuol trovare uno zero
% f1: derivata prima della funzione
%  imax: numero massimo di iterazioni consentite
%  tolx: tolleranza assoluta sul valore dello zero
%  rtolx: tolleranza relativa sul valore dello zero
%
%Questo metodo restituisce:
%  x: zero della funzione
```

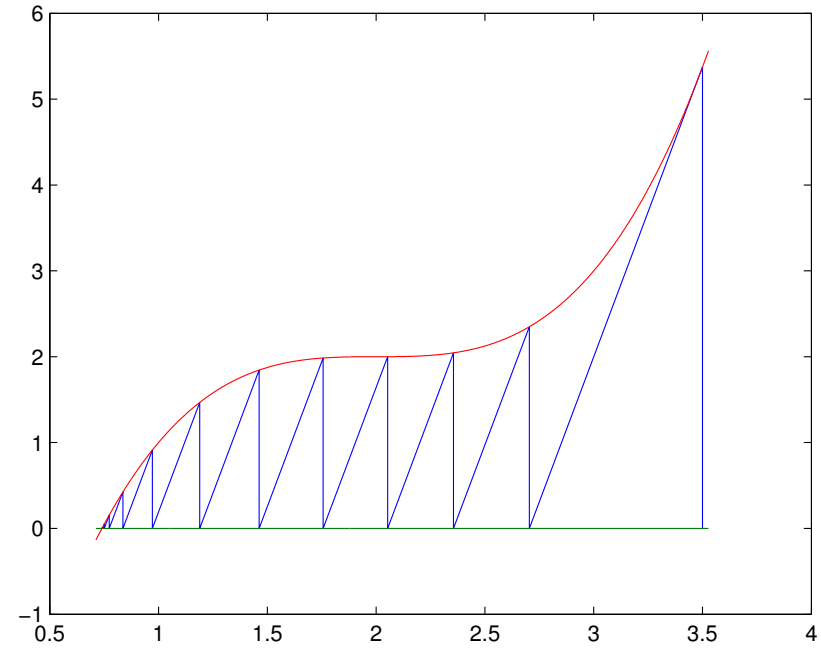

Figura 2.6: Esempio di funzionamento del metodo delle corde che ne mostra il comportamento nella determinazione dello zero della funzione $(x-2)^3+2$ con punto di innesco $x_0 = 3.5$.

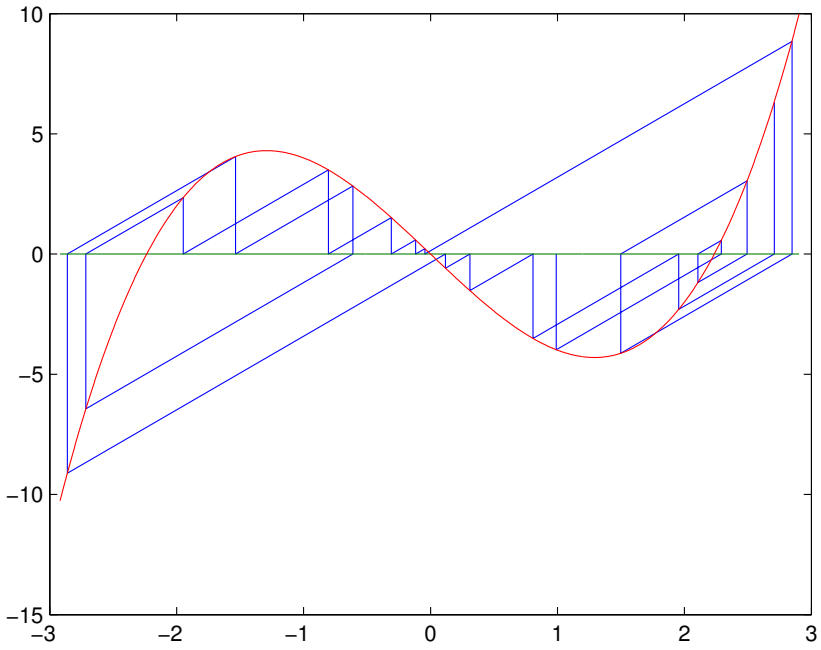

Figura 2.7: Le prime venti iterazioni del metodo delle corde per trovare una radice di $x^3 - 5x$ con punto di innesco $x_0 = 0.99$: in questo caso il metodo converge molto lentamente e può girare per più di 100000 iterazioni senza avvicinarsi ad una delle radici con una tolleranza di 10^-6

```
%  i: numero di iterazioni fatte
%

function [x,i] = secanti(f,f1,x0,itmax,tolx,rtolx)
   i=0;
   fx = feval(f,x0);
   f1x = feval(f1,x0);

   if f1x == 0
       if fx==0
           return
       end
      error('La derivata prima ha assunto valore zero, impossibile continuare!')
   end

   x = x0 - fx/f1x;

   while (i<itmax) & ((abs(x-x0)/(tolx+rtolx*(abs(x))))>1)
       i = i+1;
       fx0 = fx;
       fx = feval(f,x);
       t = (fx-fx0);
    %Questo controllo serve ad aggirare il problema della cancellazione numerica
    %in quanto la sottrazione appena fatta è mal condizionata (stiamo sottraendo
    %due valori molto vicini). In caso t sia 0 non possiamo proseguire, ma
    %vale la pena controllare di aver trovato una soluzione. Il controllo
    %sulla tolleranza delle ascisse non ha senso in quanto sarebbe anch'esso
    %soggetto alla cancellazione numerica e quindi inaffidabile.
       if t == 0
         if fx == 0
             return
         elseif ((abs(x-x0)/(tolx+rtolx*(abs(x))))<=1)
            return
         end
        error('Impossibile determinare la radice nella tolleranza desiderata')
    end
       x1 = (fx*x0-fx0*x)/t;
       x0 = x;
       x = x1;
   end
%Se il ciclo sopra e' terminato perche' abbiamo raggiunto il limite massimo di
%iterazioni, non abbiamo ottenuto un risultato accettabile.
   if ((abs(x-x0)/(tolx+rtolx*(abs(x))))>1), disp('Il metodo non converge'), end
```

Costo computazionale

Questo metodo ad ogni ciclo iterativo svolge soltanto 5 operazioni floating point ed una valutazione di funzione, il costo per iterata è quindi veramente molto basso.

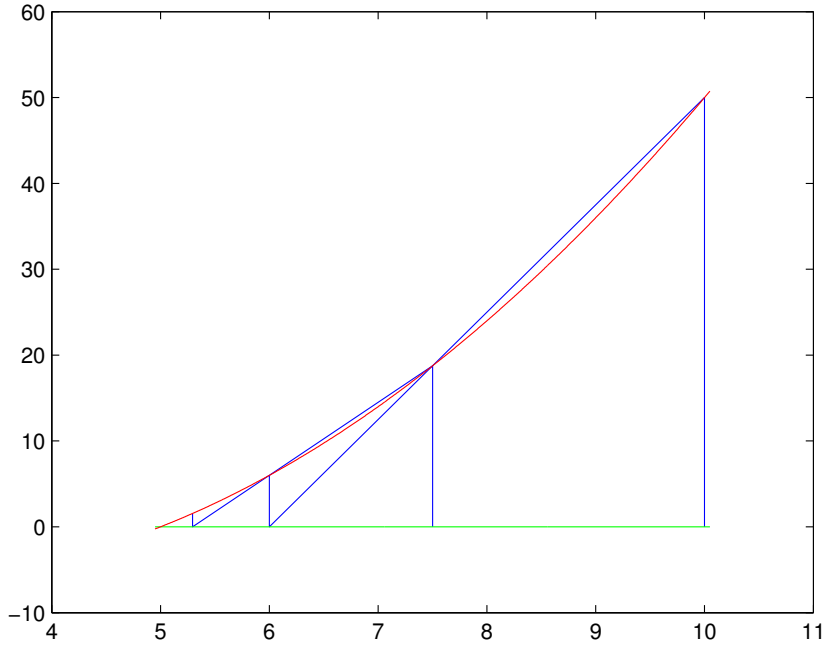

Figura 2.8: Esempio di funzionamento del metodo delle secanti che ne mostra il comportamento nella determinazione dello zero della funzione $x^2 - 5x$ con punto di innesco $x_0 = 10$.

Confronto tra il metodo di Newton, metodo di Newton nel caso di radici di molteplicità nota e metodo di accelerazione di Aitken

Queste tabelle mostrano il comportamento di questi tre metodi nella valutazione delle funzioni:

$$f_1(x) = (x-1)^{10}, \quad f_(x) = (x-1)^{10}e^x$$

per valori decrescenti della tolleranza tolx e punto iniziale $x_0 = 10$.

$f_1(x)$						$f_2(x)$
$tolx$	i	x		$tolx$	i	x
NTN						
10^{-2}	43	1.08727963568088		10^{-2}	52	1.08957265092411
10^{-4}	87	1.00084641497829		10^{-4}	96	1.00087725269306
10^{-6}	131	1.00000820831010		10^{-6}	140	1.00000850818739
10^{-8}	174	1.00000008844671		10^{-8}	184	1.00000008251024
10^{-10}	218	1.00000000085773		10^{-10}	227	1.00000000088907
NTM						
10^{-2}	1	1		10^{-2}	4	1.00000041794715
10^{-4}	1	1		10^{-4}	5	1.00000000000002
10^{-6}	1	1		10^{-6}	5	1.00000000000002
10^{-8}	1	1		10^{-8}	6	1
10^{-10}	1	1		10^{-10}	6	1
ATK						
10^{-2}	2	1		10^{-2}	6	0.99999998332890
10^{-4}	2	1		10^{-4}	7	1
10^{-6}	2	1		10^{-6}	7	1
10^{-8}	2	1		10^{-8}	7	1
10^{-10}	2	1		10^{-10}	7	1

Confronto tra il metodo di Newton, metodo delle corde, metodo delle secanti e metodo di bisezione

Questa tabella mostra il comportamento di questi metodi nella valutazione della funzione:

$$f(x) = x - cos(x)$$

per valori decrescenti della tolleranza tolx e punto iniziale $x_0 = 0$, mentre per il metodo di bisezione consideriamo l'intervallo $[0, 1]$.

$f(x)$		
$tolx$	i	x
NTN		
10^{-2}	3	0.73908513338528
10^{-4}	3	0.73908513338528
10^{-6}	4	0.73908513321516
10^{-8}	4	0.73908513321516
10^{-10}	5	0.73908513321516
SCT		
10^{-2}	3.	0.73911936191163
10^{-4}	4	0.73908511212746
10^{-6}	5	0.73908513321500
10^{-8}	6	0.73908513321516
10^{-10}	6	0.73908513321516
CRD		
10^{-2}	12	0.73560474043635
10^{-4}	24	0.73905479074692
10^{-6}	35	0.73908552636192
10^{-8}	47	0.73908513664657
10^{-10}	59	0.73908513324511
BIS		
10^{-2}	7	0.734375
10^{-4}	13	0.739013671875
10^{-6}	20	0.73908424377441
10^{-8}	25	0.73908513784409
10^{-10}	31	0.73908513318747

Capitolo 3

Risoluzione di sistemi lineari

Un sistema lineare può essere scritto nella forma:

$$A\mathbf{x} = \mathbf{b}$$

dove $A = (a_{ij}) \in \mathbb{R}^{m \times n}$ matrice dei coefficienti, $\mathbf{b} = (b_i) \in \mathbb{R}^m$ vettore dei termini noti e $\mathbf{x} = (x_i) \in \mathbb{R}^n$ vettore delle incognite. Assumeremo che la matrice abbia un numero di righe maggiore o uguale a quello delle colonne ($m \geq n$) e che abbia rango massimo ($rank(A) = n$); queste assunzioni riescono comunque a coprire una buona parte dei problemi derivati dalle applicazioni.

3.1 Matrici quadrate

Iniziamo esaminando il caso in cui $m = n$, ovvero A è una matrice quadrata. In quanto $rank(A) = n$, A è una matrice nonsisngolare e la soluzione del sistame esiste ed è unica:

$$\mathbf{x} = A^{-1}\mathbf{b} \quad \text{con} \quad AA^{-1} = A^{-1}A = I$$

3.1.1 Matrici diagonali

Analizziamo adesso il caso in cui A sia una matrice diagonale, ovvero:

$$\begin{pmatrix} a_{11} & & 0 \\ & \ddots & \\ 0 & & a_{nn} \end{pmatrix}$$

in cui valgono le proprietà:

- $a_{ij} = 0 \Leftrightarrow i \neq j$.
- $det(A) = \prod_{i=1}^{n} a_{ii} \neq 0 \Longrightarrow a_{ii} \neq 0 \quad \forall i, 0 \leq i \leq n$.

In questo caso il sistema lineare $A\mathbf{x} = \mathbf{b}$ assume la forma:

$$\begin{aligned} a_{11}x_1 &= b_1 \\ &\vdots \\ a_{nn}x_n &= b_n \end{aligned}$$

e quindi la soluzione ottenibile:

$$x_i = \frac{b_i}{a_{ii}}, \quad i = 1, ..., n.$$

è ben posta in quanto come detto prima $a_{ii} \neq 0$ per $i = 1, ..., n$. Il costo di questo metodo sia in termini di occupazione di memeoria che di quantità di operazioni floating-point è lineare con la dimensione del problema; sono sufficiendi n operazioni floating-point ed un vettore di dimensione n per memorizzare gli elementi della diagonale della matrice A.

3.1.2 Matrici triangolari

Le matrici triangolari, che contengono informazione soltanto in una porzione triangolare, possono essere di due tipi:

- Triangolari inferiori: $a_{ij} = 0$ se $j > i$

$$A = \begin{pmatrix} a_{11} & & \\ \vdots & \ddots & \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}$$

- Triangolari superiori: $a_{ij} = 0$ se $i > j$

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ & \ddots & \vdots \\ & & a_{nn} \end{pmatrix}$$

Nel caso in cui la matrice A sia triangolare inferiore (il caso in cui sia triangolare superiore è analogo) il sistema lineare è della forma:

$$\begin{array}{lll} a_{11}x_1 & = & b_1, \\ a_{21}x_1 + a_{22}x_2 & = & b_2, \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 & = & b_3, \\ \vdots \qquad\qquad \ddots & \vdots & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n & = & b_n, \end{array}$$

e quindi le soluzioni sono ottenibili tramie sostituzioni successive in avanti:

$$\begin{array}{lll} x_1 & = & b_1/a_{11} \\ x_2 & = & (b_2 - a_{21}x_1)/a_{22} \\ x_3 & = & (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33} \\ & \vdots & \\ x_n & = & (b_n - \sum_{j=1}^{n-1} a_{nj}x_j)/a_{nn} \end{array}$$

In quanto A è nonsingolare, si ha che $a_{ii} \neq 0, \quad i = 1, ..., n$ e quindi tutte le operazioni sopra descritte sono ben definite. In quanto la matrice contiene informazione solo in una sua porzione triangolare, è sufficiente memorizzare tale porzione che contiene:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$$

posizioni di memoria. Per quanto riguarda il costo computazionale invece, da quanto detto sopra si ha che sono necessari 1flop per calcolare x_1,3flop per calcolare x_2,...,$2n-1$flop per calcolare x_n, e quindi:

$$\sum_{i=1}^{n}(2i - 1) = n^2 \text{ flop.}$$

3.1.3 Matrici ortogonali

Una matrice è ortogonale se e solo se vale la proprietà: $A^{-1} = A^T$. In questo caso quindi la soluzione del sistema lineare è ottenibile facilmente calcolando $\mathbf{x} = A^T\mathbf{b}$, al costo di un prodotto matrice-vettore; saranno richiesti pertanto circa $2n^2$flops ed n^2 posizioni di memoria.

3.1.4 Proprietà

Mostriamo adesso alcune proprietà interessanti di questi tipi di matrici che andremo ad utilizzare con i metodi di fattorizzazione.

Teorema

La somma ed il prodotto di matrici triangolari inferiori (superiori) è ancora una matrice triangolare inferiore (superiore).

Dimostrazione:

Dimostriamo il caso con matrici triangolari inferiori, l'altro si domostra in maniera analoga.
Siano L_1, L_2 matrici triangolari inferiori, ed indichiamo con (a_{ij}) e $(b_{ij}$ i corrispettivi elementi.
Per ipotesi abbiamo che $a_{ij} = b_{ij} = 0, \quad j > i$.
Allora gli elementi della matrice somma saranno composti:
$L_1 + L_2 = (a_{ij} + b_{ij})$ ma se $j > i$ si ha $(a_{ij} + b_{ij}) = 0 + 0 = 0$, ovvero $L_1 + L_2$ è triangolare inferiore.
Vediamo adesso il caso del prodotto.
Chiamaiamo c_{ij} gli elementi della matrice L_1L_2. Vogliamo dimostrare che $c_{ij} = 0$, se $j > i$.
L'elemento c_{ij}, quando $j > i$, è scrivibile nella forma: $\mathbf{e}_i^T(L_1L_2)\mathbf{e}_j$. Svolgendo le operazioni in quest'ordine: $(\mathbf{e}_i^T L_1)(L_2\mathbf{e}_j)$ si ha:

$$\begin{pmatrix} a_{i1}a_{i2}...a_{ii}\overbrace{0...0}^{n-i} \end{pmatrix} \quad \begin{pmatrix} 0 \\ \vdots \\ 0 \\ b_{ij} \\ \vdots \\ b_{nj} \end{pmatrix}$$

ed il vettore b contiene 0 nelle prime $j-1$ posizioni. Ma $j > i$ e quindi i primi j zero del vettore b annullano tutti i termini diversi da zero del vettore a, e $c_{ij} = 0$.

Teorema

Se $L_1 = (a_{ij}) e L_2 = (b_{ij})$ sono matrici triangolari inferiori, allora gli elementi diagonali della matrice L_1L_2 sono dati da $(a_{ii}b_{ii})$.
Dimostrazione:
Sia c_{ij} elemento della matrice risultante. Per definizione di prodotto tra matrici, si ha che

$$c_{ij} = \sum_{k=1}^{n}(a_{ik}b_{kj})$$

Ricordando che $a_{ij} = b_{ij} = 0$ se $j > i$, si nota facilmente che la somma non è interessata elementi che coinvolgono valori di k minori di j o maggiori di i in quanto il prodotto è annullato dall'elemento rispettivamente della matrice L_2 o L_1; l'unico prodotto che non si annulla quando $i = j$ (stiamo calcolando la diagonale) si ha con $k = i = j$ e coinvolge appunto solamente i fattori a_{kk} e b_{kk}. Cvd.

Teorema

La somma ed il prodotto di matrici triangolari inferiori (o superiori) è ancora una matrice dello stesso tipo.
Dimostrazione:
Siano L ed S due matrici tiangolari inferiori. Se $i < j$ quindi $L(i,j) = S(i,j) = 0$, e sommandole, si ha quindi che $L + S(i,j) = L(i,j) + S(i,j) = 0 + 0 = 0$, ovvero anche la loro somma è una matrice triangolare inferiore. Nel caso del prodotto si ha che

$$L \cdot S(i,j) = \sum_{k=1}^{n} L(i,k) \cdot S(k,j)$$

ma se $i < j$, $L(i,j) = S(i,j) = 0$ e quindi $S(k,j) = 0$ per $k < j$ e $L(i,k) = 0$ per $, k > i$ e tutti i prodotti si annullano. Lo stesso ragionamento vale per matrici triangolari superiori, con $i > j$.

Lemma

Il prodotto di due matrici triangolari inferiori (o superiori) a diagonale unitaria è ancora una matrice dello stesso tipo.
Dimostrazione:
Abbiamo appena dimostrato che il prodotto conserva la triangolarità, vediamo cosa succede alla diagonale:

$$L \cdot S(i,i) = \sum_{k=1}^{n} L(i,k) \cdot S(k,i)$$

ma in questo caso l'unico elemento non nullo nella sommatoria è $L(1,1) \cdot S(1,1,) = 1$.

Lemma

Se L è nonsingolare e triangolare allora L^{-1} è triangolare dello stesso tipo.
Dimostrazione:
Supponiamo L triangolare inferiore e L^{-1} triangolare superiore. Allora si avrebbe che $(L \cdot L^{-1})(1,2) = L(1,1) * L^{-1}(1,1) + 0 + 0 \neq 0$, in quanto per ipotesi $L(1,1), L^{-1}(1,1) \neq 0$. Ne consegue risultato non può essere la matrice identità, e l'inversa di una matrice triangolare deve pertanto essere anch'essa triangolare dello stesso tipo.

Implementazione

Questo codice implementa gli algoritmi di risoluzione dei sistemi lineari di tipo semplice, quali i triangolari inferiori e superiori, diagonale, ortogonale e bidiagonale superiore e inferiore.

```
%Questo metodo risolve sistemi lineari semplici.
%
% b = sisem(A,b,type)
%
```

```
%Questo metodo prende in input:
%   A : Matrice quadrata
%   b : Vettore dei termini noti
%   type : Stringa di valore (inf | sup | dia | ort | sbd | ibd) che identifica
%     il tipo di matrice, rispettivamente:
%               -   inf : matrice triangolare inferiore
%               -   sup : matrice triangolare superiore
%               -   dia : matrice diagonale
%               -   ort : matrice ortgonale
%               -   sbd : matrice bidiagonale superiore
%               -   ibd : matrice bidiagonale inferiore
%
%E restituisce:
% b: vettore delle soluzioni del sistema.
%


function b = sisem(A,b,type)

if type == 'inf'
    n = size(b);
    for j = 1:n

if A(j,j)~=0
        b(j,:) = b(j,:)./A(j,j);
else error('Matrice singolare'), end

    for i = j+1:n
            b(i,:) = b(i,:)-A(i,j).*b(j,:);
    end

    end

elseif type == 'sup'
    n = size(b);
    for j = n:-1:1

    if A(j,j)~=0
        b(j,:) = b(j,:)./A(j,j);
else error('Matrice singolare'), end

    for i = 1:j-1
            b(i,:) = b(i,:)-A(i,j).*b(j,:);
    end

    end

elseif type == 'dia'
    n = size(b);
    for i=1:n
        if b(i)~=0
            b(i,:)=A(i,i)./b(i,:);
        else error('Matrice singolare'), end
    end
```

```
elseif type == 'ort'
    b = A'*b;

elseif type == 'sbd'
    n = size(b);
    for j = n:-1:1
        if A(j,j)~=0
            b(j,:) = b(j,:)./A(j,j);
        else error('Matrice singolare'), end
        if j-1 >=1
            b(j-1,:) = b(j-1,:)-A(j-1,j).*b(j,:);
        end
    end

elseif type == 'ibd'
    n = size(b);
    for j = 1:n-1
        if A(j,j)~=0
            b(j,:) = b(j,:)./A(j,j);
        else error('Matrice singolare'), end
        b(j+1,:) = b(j+1,:)-A(j+1,j).*b(j,:);
    end
    if A(j+1,j+1)~=0
    b(j+1,:) = b(j+1,:)./A(j+1,j+1);
    else error('Matrice singolare'), end

else
    error('Type non corretto.')
end
```

3.2 Metodi di fattorizzazione

Da quanto descritto fino ad ora si può concludere che siamo in grado di trattare in maniera efficiente matrici particolari quali quelle diagonali e quelle triangolari. Ovviamente non tutte le matrici sono di questi tipi, ma siamo in grado di definire alcuni metodi, detti di fattorizzazione, che permettodo di descrivere una matrice generica come prodotto di *fattori* che siano matrici particolari, ovvero del tipo a noi più congeniale. Vogliamo quindi poter scrivere $A = F_1F_2...F_k$,dove $F_k \in \mathbb{R}^{n\times n}$ sono matrici diagonali, triangolari oppure ortogonali. In questo modo, tenendo conto della fattorizzazione di A, è possibile risolvere il sistema $A\mathbf{x} = \mathbf{b}$ nel seguente modo:

$$F_1\mathbf{x}_1 = \mathbf{b}, \quad , F_2\mathbf{x}_2 = \mathbf{x}_1, \quad ... \quad , F_k\mathbf{x}_k = \mathbf{x}_{k-1}, \quad \mathbf{x} \equiv \mathbf{x}_k$$

É importante notare che tutti i sistemi appena citati sono di tipo semplice e quindi immediatamente risolvibili, e che non è necessario memorizzare tutte le soluzioni intermedie, perché queste perdono di utilità una volta calcolata quella successiva.

3.3 Fattorizzazione LU

In base a quanto detto fin'ora, una situazione interessante si ha quando stiamo lavorando su di una matrice A che può essere scritta come il prodotto di due matrici particolari

che noi chiameremo Lower ed Upper, tali che L sia triangolare inferiore a diagonale unitaria e U triangolare superiore; ovvero la matrice A è *fattorizzabile* LU.

Teorema

Se A è una matrice nonsingolare e la fattorizzazione $A = LU$ esiste , allora questa è unica.
Dimostrazione:
Siano $A = L_1U_1 = L_2U_2$ due fattorizzazioni di A; si ha che:

$$0 \neq det(A) = det(L_2U_2) = det(L_2)det(U_2) = det(U_2).$$

U_2 è quindi nonsingolare e si ha che $L_1^{-1}L_2 = U_1U_2^{-1} \equiv D$. Ma $L_1^{-1}L_2$ e $U_1U_2^{-1}$ sono rispettivamente triangolare inferiore e triangolare superiore, pertanto D deve essere diagonale. Ma $L_1^{-1}L_2$ è una matrice a diagonale unitaria e questo implica che D sia la matrice Identità; ne consegue quindi che $L_1 = L_2$ e $U_1 = U_2$. cvd.
Rimane adesso da stabilire l'esistenza di tale fattorizzazione, e procederemo illustrando un metodo di fattorizzazione.
Sia $\mathbf{v} = (v_1...v_n)^T \in \mathbb{R}^n$ vettore tale che $v_k \neq 0$, si supponga di volerne azzerare tutte le componenti successive alla k-esima esclusa, lasciando invariate tutte le altre, moltiplicandolo a sinistra con una matrice triangolare inferiore a diagonale unitaria $L \in \mathbb{R}^{n\times n}$. É possibile definire il *vettore elementare di Gauss* come:

$$\mathbf{g} = \frac{1}{v_k}(\overbrace{0...0}^{k}, v_{k+1}, ..., v_n)^T$$

grazie al fatto di avere $v_k \neq 0$, e quindi la *matrice elementare di Gauss*:

$$L \equiv I - \mathbf{g}\mathbf{e}_k^T.$$

Ovvero:

$$L = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & -\frac{v_{k+1}}{v_k} & \ddots & \\ & & \vdots & & \ddots \\ & & -\frac{v_n}{v_k} & & 1 \end{pmatrix},$$

$$L\mathbf{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_k \\ 0 \\ \vdots \\ 0 \end{pmatrix} = (I - \mathbf{g}\mathbf{e}_k^T)\mathbf{v} = I\mathbf{v} - \mathbf{g}\overbrace{(\mathbf{e}_k^T\mathbf{v})}^{\mathbf{v}_k} = \mathbf{v} - \mathbf{g}\mathbf{v}_k$$

Questa matrice opera come ci eravamo preposti, infatti le prime k componenti del vettore $L\mathbf{v}$ coincidono con le rispettive di $\mathbf{v}$.
L'inversa di L è facilmente ottenibile come: $L^{-1} = I + \mathbf{g}\mathbf{e}_k^T$, infatti:

$$L^{-1}L = (I + \mathbf{g}\mathbf{e}_k^T)(I - \mathbf{g}\mathbf{e}_k^T) = I - \mathbf{g}\mathbf{e}_k^T + \mathbf{g}\mathbf{e}_k^T - \mathbf{g}(\mathbf{e}_k^T\mathbf{g})\mathbf{e}_k^T = I.$$

in quando $\mathbf{e}_k^T\mathbf{g}$,k-esima componente del vettore $\mathbf{g}$, è nulla.
Procediamo adesso alla trasformazione della matrice nonsingolare A in una matrice

triangolare superiore mediante la procedura iterativa denominata *metodo di eliminazione di Gauss*, che dura $n-1$ passi.
Definiamo:

$$A \equiv A^{(1)} = \begin{pmatrix} a_{11}^{(1)} & \cdots & a_{1n}^{(1)} \\ \vdots & & \vdots \\ a_{n1}^{(1)} & \cdots & a_{nn}^{(1)} \end{pmatrix}$$

la matrice al primo passo. L'indice superiore di ciascun elemento tiene traccia del passo più recente in cui il valore dell'elemento è stato modificato dalla procedura. Vogliamo quindi come primo passo definire una matrice triangolare inferiore a diagonale unitaria che sia in grado di trasformare A rendendo la prima colonna strutturalmente uguale alla relativa di una matrice triangolare superiore.
Se $a_{11}^{(1)} \neq 0$, possiamo definire il *primo* vettore elementare di Gauss:

$$g_1 \equiv \frac{1}{a_{11}^{(1)}}(0, a_{21}^{(1)}, ..., a_{n1}^{(1)})^T$$

e la *prima* matrice elementare di Gauss:

$$L_1 \equiv I - \mathbf{g}_1 \mathbf{e}_1^T = \begin{pmatrix} 1 & & & \\ -\frac{a_{21}^{(1)}}{a_{11}^{(1)}} & 1 & & \\ \vdots & & \ddots & \\ -\frac{a_{n1}^{(1)}}{a_{11}^{(1)}} & & & 1 \end{pmatrix}$$

per avere:

$$L_1 A = \begin{pmatrix} a_{11}^{(1)} & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \vdots & & \vdots \\ 0 & a_{n2}^{(2)} & \cdots & a_{nn}^{(2)} \end{pmatrix} \equiv A^{(2)}.$$

Possiamo osservare come, per la struttura di L, la prima riga del prodotto $L_1 A$ non venga modificata.
Generalizziamo quindi lo stesso procedimento al passo i-esimo, tenendo conto che $a_{jj}^{(j)} \neq 0$ per ogni $j < i$:

$$L_{i-1}...L_2 L_1 A = \begin{pmatrix} a_{11}^{(1)} & \cdots & \cdots & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & \ddots & & & & \vdots \\ \vdots & \ddots & a_{i-1,i-1}^{(i-1)} & \cdots & \cdots & a_{i-1,n}^{(i-1)} \\ \vdots & & 0 & a_{ii}^{(i)} & \cdots & a_{in}^{(i)} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a_{ni}^{(i)} & \cdots & a_{nn}^{(i)} \end{pmatrix} \equiv A^{(i)}$$

A questo punto solo se $a_{ii}^{(i)} \neq 0$ sarà possibile definire l'i-esimo vettore di Gauss:

$$g_i \equiv \frac{1}{a_{ii}^{(i)}}(\underbrace{0...0}_{i}, a_{i+1,i}^{(i)}, ..., a_{ni}^{(i)})^T,$$

e quindi la i-esima matrice elementare di Gauss:

$$L_i \equiv I - \mathbf{g}_i \mathbf{e}_i^T = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -\frac{a_{i+1,i}^{(i)}}{a_{ii}^{(i)}} & \ddots & & \\ & & \vdots & & \ddots & \\ & & -\frac{a_{ni}^{(i)}}{a_{ii}^{(i)}} & & & 1 \end{pmatrix}$$

e quindi:

$$L_i A^{(i)} = L_i ... L_2 L_1 A = \begin{pmatrix} a_{11}^{(1)} & \cdots & \cdots & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & \ddots & & & & \vdots \\ \vdots & \ddots & a_{ii}^{(i)} & \cdots & \cdots & a_{in}^{(i)} \\ \vdots & & 0 & a_{i+1,i+1}^{(i+1)} & \cdots & a_{i+1,n}^{(i+1)} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a_{n,i+1}^{(i+1)} & \cdots & a_{nn}^{(i+1)} \end{pmatrix} \equiv A^{(i+1)}$$

Se si osserva che nella i-esima matrice di Gauss ($L_i \equiv I - \mathbf{g}_i \mathbf{e}_i^T$) le prime i righe corrispondono con quelle della matrice identità, si può notare che le prime i righe delle matrici $A^{(i)}$ e $A^{(i+1)}$ coincidono e gli elementi azzerati al passo i-esimo (che stanno nelle prime $i-1$ colonne) non vengono alterati dalla moltiplicazione per L_i.
Se risulta possibile iterare i passi fin qui descritti $i = n-1$ volte, si ottiene che:

$$A^{(n)} \equiv U \equiv L_{n-1} ... L_1 A = \begin{pmatrix} a_{11}^{(1)} & \cdots & \cdots & a_{1n}^{(1)} \\ & \ddots & & \vdots \\ & & a_{n-1,n-1}^{(n-1)} & a_{n-1,n}^{(n-1)} \\ & & & a_{nn}^{(n)} \end{pmatrix}$$

Adesso, osservando che la matrice $L_{n-1}...L_1$ è il prodotto di matrici triangolari inferiori a diagonale unitaria, è anch'essa di questo tipo, e così sarà anche la sua inversa. Tenendo conto che:

$$\mathbf{e}_k^T \mathbf{g}_i \equiv g_{ki} = 0, \qquad \text{per } k \leq i,$$

e ponendo formalmente la matrice $L_{n-1}...L_1$ uguale a L^{-1}:

$$\begin{aligned} L &= (L_{n-1}...L_1)^{-1} = L_1^{-1}...L_{n-1}^{-1} \\ &= (I + \mathbf{g}_1 \mathbf{e}_1^T)...(I + \mathbf{g}_{n-1} \mathbf{e}_{n-1}^T) = I + \mathbf{g}_1 \mathbf{e}_1^T + ... + \mathbf{g}_{n-1} \mathbf{e}_{n-1}^T \\ &= \begin{pmatrix} 1 & & & \\ g_{12} & 1 & & \\ \vdots & \ddots & \ddots & \\ g_{n1} & \cdots & g_{n,n-1} & 1 \end{pmatrix} \end{aligned}$$

Siamo quindi riusciti ad ottenere due matrici L ed U tali che $A = LU$.
Consideriamo adesso le condizioni sufficienti a garantire l'esistenza della fattorizzazione LU: il metodo di eliminazione di Gauss è definito se e solo se è possibile costruire la matrice di Gauss al generico passo i-esimo.

Lemma

Se A è una matrice nonsingolare, la fattorizzazione $A = LU$ è definita se e soltanto se $a_{ii}^{(i)} \neq 0, \quad i = 1, 2, ..., n$, ovvero se e solo se la matrice U è nonsingolare.

Definizione

Si dice sottomatrice principale di ordine k di una generica matrice $A \in \mathbb{R}^{n \times n}$ l'intersezione tra le sue prime k righe e k colonne, ovvero:

$$A_k = \begin{pmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & & \vdots \\ a_{k1} & \cdots & a_{kk} \end{pmatrix}$$

In particolare si ha che il determinante $det(A_k)$ è il *minore principale di ordine* k di A.

Lemma

Una matrice triangolare è nonsingolare se e solo se tutti i suoi minori principali sono non nulli.

Lemma

Il minore di ordinde k di A in $A = LU$ coincide con il minore di ordine k di U in $A^{(n)} \equiv U$

Teorema

Se A è una matrice nonsingolare, la fattorizzazione $A = LU$ esiste se e solo se tutti i minori principali di A sono non nulli.

La fattorizzazione quindi esiste se e solo se U è nonsisingolare, e questo equivale a richiedere che tutti i minori principali di U siano non nulli, e quindi anche tutti quelli di A devono essere non nulli.

3.3.1 Costo computazionale

Esaminiamo adesso il costo dell'algoritmo di eliminazione di Gauss in termini di occupazione di memoria.
Strutturalmente, al passo i-esimo l'algoritmo azzera, nella matrice $A^{(1)}$, le componenti $i+1, ..., n$ della colonna i; al contempo si ottiene l'i-esimo vettore di Gauss che contiene informazioni soltanto nelle ultime $n - i$ componenti. Possiamo quindi pensare di sovrascrivere A durante il processo, ottenendo una matrice che contiene la parte significativa della matrice U nella porzione triangolare superiore e le componenti significative della matrice L (ovvero gli elementi significativi dei vettori di Gauss) nella parte strettamente inferiore, tenendo conto che la diagonale della matrice L ha tutti elementi pari a 1 e non necessita quindi di essere memorizzata esplicitamente. Possiamo quindi concludere che il metodo di eliminazione di Gauss non richiede memoria addizionale alla matrice di partenza A, che è quanto di meglio si possa richiedere.
Per quanto riguarda invece il costo computazionale, ad ogni passo i dell'algoritmo è necessario calcolare:

- l'i-esimo vettore di Gauss
- il prodotto $L_i A^{(i)} = (I - \mathbf{g}_i \mathbf{e}_i^T) A^{(i)} = A^{(1)} - \mathbf{g}_i (\mathbf{e}_i^T A^{(i)}) \equiv A^{(i+1)}$.

Considerando che le prime i componenti del vettore $\mathbf{g}_i$ sono nulle, che il vettore $\mathbf{e}_i^T A^{(i)}$ corrisponde alla i-esima riga di $A^{(i)}$ che ha le prime i componenti nulle e che sappiamo a priori che le componenti da $(i+1)an$ della colonna i della matrice $A^{(i+1)}$ sono nulle, si nota che solo la sottomatrice delimitata da $(i+1, i+1), (n, n)$ verrà modificata al passo i. Il costo computazionale di una iterazione consiste in $n-i$ divisioni per calcolare il vettore di Gauss, $(n-i)^2$ sottrazioni ed altrettante moltiplicazioni per aggiornare la matrice; in totale la fattorizzazione avrà un costo approssimativo di:

$$\sum_{i=1}^{n}(2(n-i)^2 + (n-i)) \approx \frac{2}{3}n^3.$$

Implementazione

Questo codice implementa l'algoritmo di fattorizzazione LU:

```
%Fattorizzazione LU
%
% [A,t]=fattLUlib(A)
%
%La funzione prende in input
% A: una matrice quadrata
%E restituisce
% A: una matrice contenente la porzione strettamente triangolare
% inferiore della matrice L e la porzione triangolare superiore della
% matrice U
% t: tempo di computazione
function [A]=fattLUlib(A)

[m,n]=length(A);
if m~=n, error('La matrice non e quadrata'), end

for i = 1:n-1

    if A(i,i)==0
        error('Matrice non fattorizzabile LU')
    end

    A(i+1:n,i) = A(i+1:n,i)/A(i,i);

    A(i+1:n,i+1:n) = A(i+1:n,i+1:n)-A(i+1:n,i)*A(i,i+1:n);
end
```

3.4 Matrici a diagonale dominante

Esistono delle classi di matrici per cui è vero che se A è nonsingolare allora lo è anche $A_k, \quad \forall k = 1, ...n$, e sono le matrici a diagonale dominante.

Definizione

La matrice $A \in \mathbb{R}^{n\times n}$ si dice *a diagonale dominante per righe* se:

$$|a_{ii}| > \sum_{j\neq i} |a_{ij}|, \quad i = 1, ..., n;$$

e *a diagonale dominante per colonne* se:

$$|a_{ii}| > \sum_{j \neq i} |a_{ji}|, \quad i = 1, ..., n;$$

Per questa classe di matrici valgono le seguenti proprietà:

Lemma

Se A è una matrice a diagonale dominante per righe (o per colonne), allora anche tutte le sue sottomatrici principali sono dello stesso tipo

Lemma

Una matrice A è a diagonale dominante per righe (o per colonne) se e solo se A^T è a diagonale dominante per colonne (per righe).

Teorema

Se una matrice A è diagonale dominante (per righe o per colonne), allora A è non singolare.
Dimostrazione:
Sia A una matrice a diagonale dominante per righe, supponiamo per assurdo che A sia singolare. Allora deve esistere $\mathbf{x} \neq \mathbf{0}$ tale che $A\mathbf{x} = \mathbf{0}$. Assumiamo

$$\mathbf{x}_k = max(|\mathbf{x}_i|) = 1, \text{ e quindi } |\frac{\mathbf{x}_i}{\mathbf{x}_k}| \leq 1 \quad \forall i = 1, ..., n.$$

Ma se $A\mathbf{x} = \mathbf{0}$ allora $\mathbf{e}_k^T A\mathbf{x} = \mathbf{e}_k^T \mathbf{0} = \mathbf{0}$, ne consegue che:

$$\sum_{j=i}^{n} a_{kj} x_j = 0 \Rightarrow a_{kk} x_k = a_{kk} = - \sum_{j=1, j \neq k}^{n} a_{kj} x_j =$$

$$= |a_{kk}| = \left| \sum_{j \neq k} a_{kj} x_j \right| \leq \sum_{j \neq k} |a_{kj} x_j| \leq \sum_{j \neq k} |a_{kj}|$$

ma allora A non è diagonale dominante. Assurdo.

Lemma

Se A è una matrice a diagonale dominante, allora A è fattorizzabile LU.

3.5 Fattorizzazione LDL^T

Un'altra classe di matrici fattorizzabili LU è quella delle matrici simmetriche e definite positive (sdp).

Definizione

Una matrice $A \in \mathbb{R}^{n \times n}$ è sdp se è simmetrica (ovvero $A = A^T$) e se vale:

$$\forall x \in \mathbb{R}^n, x \neq 0 : \quad \mathbf{x}^T A\mathbf{x} > 0.$$

Lemma

Se A è sdp allora $a_{ii} > 0, \quad \forall i = 1, ..., n.$
Dimostrazione:
Infati, se A è sdp si dee avere che $a_{ii} = \mathbf{e}_i^T A \mathbf{e}_i > 0$. Cvd.

Lemma

Se A è sdp allora A è nonsingolare.
Dimostrazione:
Supponiamo per assurdo che A sia contemporaneamente sdp e singolare. Essendo A singolare deve esistere $\mathbf{x} \neq 0$ tale che $A\mathbf{x} = 0$, ovvero $\mathbf{x}^T A\mathbf{x} = 0$ ovvero A non è sdp. Cvd.

Teorema

A è sdp se e solo se $\forall k = 1, ..., n$, A_k è sdp.
Dimostrazione:
Procediamo per assurdo, supponendo A_k non sdp. Allora deve esistere $\mathbf{y} \in \mathbb{R}^k, \mathbf{y} \neq 0$, tale che $\mathbf{y}^T A_k \mathbf{y} \leq 0$. Allora deve esistere anche $\mathbf{x} \in \mathbb{R}^n$ tale che $x = (\mathbf{y}, 0..0)^T \neq 0$, che soddisfi $0 < \mathbf{x}^T A\mathbf{x}$ in quanto A è sdp per ipotesi. Ma allora avremmo:

$$\mathbf{x}^T A\mathbf{x} = (\mathbf{y}\overbrace{0...0}^{n-k})^T \left(\begin{array}{c|c} A_k & B^T \\ \hline \mathsf{B} & \mathsf{D} \end{array}\right) \begin{pmatrix} \mathbf{y} \\ \mathbf{0} \end{pmatrix} = \mathbf{y}^T A_k \mathbf{y} \leq 0$$

che va contro l'ipotesi.

Corollario

Se A è sdp allora A è fattorizzabile LU
Dimostrazione:
Da quanto dimostrato fino ad ora sappiamo che tutte le sottomatrici principali di A sono sdp, e quindi anche i corrispondenti minori principali sono tutti non nulli. Cvd.

Teorema

Gli elementi diagonali di una matrice simmetrica e definita positiva sono positivi.
Dimostrazione:
Se A è sdp, per definizione si ha che $a_{ii} = \mathbf{e}_i^T A \mathbf{e}_i > 0$.

Teorema

Una matrice A è sdp se e solo se è scrivibile come:

$$A = LDL^T$$

dove L è triangolare inferiore a diagonale unitaria e D è diagonale con elementi diagonali positivi.
Dimostrazione:
Per prima cosa dimostriamo che se A è scrivibile come LDL^T allora è sdp. Dobbiamo quindi dimostrare che $\forall \mathbf{x} \neq \mathbf{0}, \mathbf{x}^T LDL^T \mathbf{x} > 0$. Possiamo considerare il vettore $\mathbf{y} = L^T \mathbf{x}$ e quindi $\mathbf{y}^T = \mathbf{x}^T L$, sicuramente non nullo in quanto L è una matrice non singolare e $\mathbf{x} \neq \mathbf{0}$ per ipotesi. Si può quindi scrivere:

$$\mathbf{y}^T D\mathbf{y} = \sum_{i=1}^{n} d_{ii} \mathbf{y}^2 > 0.$$

Adesso possiamo dimostrare che se A è sdp allora $A = LDL^T$. Abbiamo dimostrato precedentemente che se A è sdp allora è fattorizzabile LU. Possiamo pensare U come $D\hat{U}$ dove:

$$D = diag(U_{11}...U_{nn}) = \begin{pmatrix} U_{11} & & 0 \\ & \ddots & \\ 0 & & U_{nn} \end{pmatrix}$$

ed

$$\hat{U} = D^{-1}U = \left(\frac{U_{ij}}{U_{ii}}\right), \quad \forall ij = i, ..., n \quad \hat{U} = \begin{pmatrix} 1 & & * \\ & \ddots & \\ 0 & & 1 \end{pmatrix}$$

ovvero D è una matrice diagonale e $\hat{U}$ triangolare superiore a diagonale unitaria. Osserviamo che

$$A^T = (LDL^T)^T = (L^T)^T D^T L^T = LDL^T = A$$

e che:

$$LU = LD\hat{U} = A = A^T = (LD\hat{U})^T = \hat{U}^T(D^T L^T)$$

ma quindi:

$$L(D\hat{U}) = \hat{U}^T DL^T \Rightarrow L = \hat{U}^T \Rightarrow \hat{U} = L^T.$$

Per l'unicità della fattorizzazione LU, in quanto $\hat{U}^T$ è triangolare inferiore a diagonale unitaria e DL^T è triangolare superiore, si ha che $\hat{U}^T = L$ e quindi la fattorizzazione $A = LDL^T$ è ben definita. Dimostriamo adesso che gli elementi diagonali di D sono positivi, facendo vedere che questa è sdp. Banalmente D è simmetrica in quanto diagonale, e, qualunque si fissi $\mathbf{x} \neq 0$ esiste un unico vettore $\mathbf{y} \neq 0$ che soddisfi $L^T\mathbf{y} = \mathbf{x}$. Si ha quindi che:

$$\mathbf{x}^T D\mathbf{x} = (L^T\mathbf{y})^T D(L^T\mathbf{y}) = \mathbf{y}^T LDL^T\mathbf{y} = \mathbf{y}^T A\mathbf{y} > 0$$

in quanto A è sdp.

3.5.1 Costo computazionale

Trattiamo innanzitutto gli aspetti che riguardano la memorizzazione delle informazioni. Per quanto riguarda la memorizzazione di matrici strettamente triangolari, dobbiamo pensare di memorizzare solo la parte che contiene informazioni, evitando di sprecare memoria per la parte strettamente triangolare nulla. Questo può essere fatto memorizzando la parte triangolare in un vettore e di utilizzare un piccolo programma che permetta di recuperare l'elemento ij della matrice triangolare dal relativo vettore. Nel caso quindi della fattorizzazione LDL^T, in quanto stiamo trattando matrici sdp (e quindi triangolari) possiamo pensare di memorizzarne solo la parte triangolare attraverso il codice appena visto, ed in più possiamo pensare di riscrivere la matrice di partenza con la porzione strettamente triangolare di L (o L^T) e la diagonale D, mantenendo quindi il costo di memorizzazione inferiore ad n^2.
Analizziamo adesso il costo in termidi di operazioni floating-point.
Stiamo lavorando su una matrice $A = (a_{ij}) = LDL^T$, una matrice $L = (l_{ij}$ che ha le

proprietà di avere $l_{ii} = 1$ ed $l_{ij} = 0$ se $j > i$ e D diagonale. Si ha quindi che:

$$\begin{aligned}
\forall i \geq j, \quad a_{ij} &= \mathbf{e}_i^T A \mathbf{e}_j = \mathbf{e}_i^T LDL^T \mathbf{e}_j \\
&= (\mathbf{e}_i^T L) D (\mathbf{e}_j^T L)^T \\
&= (\underbrace{\mathbf{e}_{11} ... \mathbf{e}_{ii}}_{i} 0...0) \begin{pmatrix} d_1 & & \\ & \ddots & \\ & & d_n \end{pmatrix} \begin{pmatrix} l_{j1} \\ \vdots \\ l_{jj} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \\
&= \sum_{k=1}^{j} l_{ik} l_{jk} d_k
\end{aligned}$$

e quindi $a_{ij} = \sum_{k=i}^{j} l_{ik} ljk d_k$, $i = j, ..., n; j = 1, ..., n$.
Si posso verificare due casi, $i = j$ e $i > j$:

$$\begin{aligned}
i = j \qquad & a_{jj} = \sum_{k=1}^{j} l_{jk}^2 d_k = l_{jj}^2 d_j + \sum_{k=1}^{j-1} l_{jk}^2 d_k \\
& \text{ma } l_{jj} = 1 \text{ quindi } d_j = a_{jj} - \sum_{k=1}^{j-1} l_{jk} l_{jk} d_k, \\
& j = 1, ..., n.
\end{aligned}$$

$$\begin{aligned}
i > j \quad & a_{ij} = \sum_{k=1}^{j} l_{ik} ljk d_k = l_{ij} l_{jj} d_j + \sum_{k=1}^{j-1} l_{ik} l_{jk} d_k \\
& l_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} d_k}{d_j}, \quad i = j+1, ..., n.
\end{aligned}$$

Si ha quindi che per effettuare il calcolo della colonna k-esima è necessaria soltanto l'informazione relativa a quelle precedenti. e che l'elemento a_{ij} sia necessario solo per il calcolo della colonna j-esima; questo ci lascia liberi di sovrascriverlo con le nuove informazioni derivanti dal calcolo appena effettuato. In particolare possiamo memorizzare le informazioni relative agli elementi d_k sulla diagonale e i fattori l_{ij} nella parte strettamente triangolare inferiore della matrice A. Si deve inoltre tenere conto che $l_{jk} d_k$ viene calcolato nel ciclo esterno e sarà poi necessario nel ciclo interno, è quindi opportuno appoggiarsi ad un array temporaneo per memorizzare il risultato di questa operazione; a questo scopo è sufficiente un array di $j-1$ locazioni di memoria in quanto l'ultimo elemento diagonale ad essere calcolato potrà essere memorizzato direttamente sulla matrice A.

Implementazione

Questo codice implementa l'algoritmo di fattorizzazione LDL^T descritto fino ad ora:

```
%Fattorizzaziole LDL^T
%
% [A] = fattLDLlib(A)
%
%Questo metodo prende in input:
% A: una matrice sdp
%
%E restituisce:
% A: matrice contenente la fattorizzazione LDL' di A.


function A = fattLDLlib(A)
```

```
n=size(A);
if A(1,1) <= 0, error('La matrice non e'' sdp'), end

A(2:n,1) = A(2:n,1)/A(1,1);

for j=2:n

    v = ( A(j,1:j-1)').*diag(A(1:j-1,1:j-1));

    A(j,j) = A(j,j)-A(j,1:j-1)*v;

    if A(j,j)<=0, error('La matrice non e'' fattorizzabile LDL'), end

    A(j+1:n,j)= (A(j+1:n,j)-A(j+1:n,1:j-1)*v)/A(j,j);

end
```

3.6 Pivoting

Prendiamo adesso in esame il caso in cui la miatrice A che vogliamo fattorizzare sia nonsingolare ma che non abbia la proprietà di avere tutti i minori principali non nulli. Abbiamo prima dimostrato che in questo caso la fattorizzazione $A = LU$ non esiste; possiamo però definire una fattorizzazione che esista sotto la sola ipotesi della nonsingolarità di A.
Supponiamo di star procedendo al primo passo del metodo di eliminazione di Gauss: per poter procedere è necessario che $a_{11} \neq 0$. Supponiamo però che $a_{11} = 0$: in quanto A è nonsingolare deve esistere, sulla prima colonna, un elemento non nullo. Possiamo quindi scegliere:

$$|a_{k_1 1}^{(1)}| \equiv max(|a_{k1}^{(1)}|) > 0, \quad k = 1, ..., n.$$

Definiamo una matrice elementare di permutazione del tipo:

$$P_1 \equiv \left(\begin{array}{ccc|c} 0 & \mathbf{0}^T & 1 & \\ \mathbf{0} & I_{k_1-2} & \mathbf{0} & \\ 1 & \mathbf{0}^T & 0 & \\ \hline & & & I_n - k_1 \end{array}\right) \equiv \left(\begin{array}{c} \mathbf{e}_k^T \\ \mathbf{e}_2^T \\ \vdots \\ \mathbf{e}_{k-1}^T \\ \mathbf{e}_1^T \\ \mathbf{e}_{k+1}^T \\ \vdots \\ \mathbf{e}_n^T \end{array}\right)$$

e non è altro che la matrice identità di ordine n con le righe 1 e k_1 scambiate. Notiamo che P_1 è una matrice simmetrica ed ortogonale, ovvero vale: $P_1 = P_1^T = P_1^{-1}$.
Segue quindi che $P_1 A^{(1)}$ non è altro che la matrice $A^{(1)}$ con le righe 1 e k_1 scambiate. Adesso abbiamo reso possibile l'effettuazione del primo passo dell'algoritmo di eliminazione di Gauss, abbiamo infatti reso non nullo a_{11}:

$$\mathbf{g}_1 = \frac{1}{a_{k_1 1}^{(1)}}(0, a_{21}^{(1)}, ..., a_{11}^{(1)}, ..., a_{n1}^{(1)})^T,$$

e la prima matrice elementare di Gauss:

$$L_1 = I - \mathbf{g}_1 \mathbf{e}_1^T$$

che consente quindi di ottenere:

$$L_1 P_1 A^{(1)} = \begin{pmatrix} a_{k_1 1}^{(1)} & \cdots & \cdots & a_{k_1 n}^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \vdots & & \vdots \\ 0 & a_{n2}^{(2)} & \cdots & a_{nn}^{(2)} \end{pmatrix} \equiv A^{(2)}$$

É impotante notare che:

$$det(A^{(2)}) = det(L_1 P_1 A) = \underbrace{det(L_1)}_{=1} \underbrace{det(P_1)}_{\pm 1} \underbrace{det(A)}_{\neq 0 \text{per ipotesi}} \neq 0$$

Ma $A^{(2)}$ è della forma:

$$\left(\begin{array}{c|c} a_{k1}^{(1)} & \mathbf{r}_2^T \\ \hline \mathbf{0} & A_2 \end{array}\right)$$

e quindi:

$$det(A^{(2)}) = det(a_{k1}^{(1)}) det(A_2) \neq 0 \Rightarrow det(A_2) \neq 0.$$

Possiamo quindi procedere in modo analogo al metodo di fattorizzazione LU, avendo al passo i-esimo la matrice:

$$L_{i-1} P_{i-1} \cdots L_1 P_1 A = \begin{pmatrix} a_{k_1 1}^{(1)} & \cdots & \cdots & \cdots & \cdots & a_{k_1 n}^{(1)} \\ 0 & \ddots & & & & \vdots \\ \vdots & \ddots & a_{k_{i-1},i-1}^{(i-1)} & \cdots & \cdots & a_{k_{i-1},n}^{(i-1)} \\ \vdots & & 0 & a_{ii}^{(i)} & \cdots & a_{in}^{(i)} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a_{n1}^{(i)} & \cdots & a_{nn}^{(i)} \end{pmatrix} \equiv A^{(i)}$$

Avremo quindi $a_{k_i i}^{(i)} \neq 0$ se A è nonsingolare. Definiamo quindi la matrice P_i che permuta le righe i e k_i (con $k_i \geq i$):

$$P_i \equiv \left(\begin{array}{c|ccc|c} I_{i-1} & & & & \\ \hline & 0 & \mathbf{0}^T & 1 & \\ & \mathbf{0} & I_{k_i - i - 1} & \mathbf{0} & \\ & 1 & \mathbf{0}^T & 0 & \\ \hline & & & & I_{n-k_i} \end{array}\right)$$

e si avrà quindi che l'elemento (i,i) della matrice $PiA^{(i)}$ diventa $a_{k_i j}^{(i)}$. Definiamo quindi l'i-esimo vettore di Gauss:

$$\mathbf{g}_i = \frac{1}{a_{k_i i}^{(i)}} (\underbrace{0, .., 0}_{i}, a_{i+1,i}^{(i)}, ..., a_{ii}^{(i)}, ..., a_{ni}^{(i)})^T,$$

e quindi l'i-esima matrice elementare di Gauss:

$$L_i = I - \mathbf{g}_i \mathbf{e}_i^T$$

tali che:

$$L_iP_iA^{(i)} = L_iP_i\cdots L_1P_1A = \begin{pmatrix} a_{k_11}^{(1)} & \cdots & \cdots & \cdots & \cdots & a_{k_1n}^{(1)} \\ 0 & \ddots & & & & \vdots \\ \vdots & \ddots & a_{k_{K_i}i}^{(i)} & \cdots & \cdots & a_{k_i,n}^{(i)} \\ \vdots & & 0 & a_{i+1,i+1}^{(i+1)} & \cdots & a_{i+1,n}^{(i+1)} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a_{n,i+1}^{(i+1)} & \cdots & a_{nn}^{(i+1)} \end{pmatrix} \equiv A^{(i+1)}$$

Con questo procedimento, se A è nonsingolare sarà sempre possibile iterare i passi appena descritti fino a $i = n-1$, ottenendo la fattorizzazione:

$$L_{n-1}P_{n-1}L_{n-1}P_{n-1}...L_1P_1A = A^{(n)} \equiv U.$$

Se si tiene conto che le matrici di permutazione elementari P_i sono sia simmetriche che ortogonali e se moltiplicate per un vettore sortiscono l'effetto di permutare le componenti i e k_i con $k_i \geq i$, si può pensare di riscrivere:

$$A^{(n)} \equiv U = \hat{L}_{n-1}\hat{L}_{n-2}...\hat{L}_1PA$$

dove

$$\begin{aligned} \hat{L}_{n-1} &\equiv L_{n-1}, \\ \hat{L}_i &\equiv P_{n-1}...P_{i+1}L_iP_{i+1}...P_{n-1}, \quad i = 1, ..., n-1, \\ P &\equiv P_{n-1}...P_1 \end{aligned}$$

P è una matrice di permutazione, ovvero una matrice ortogonale che se moltiplicata per un vettore ne permuta le componenti. Per come abbiamo definito P_i:

$$\mathbf{e}_i^TP_j = \mathbf{e}_i^T, \text{ con } j > i,$$

e quindi:

$$\begin{aligned} \hat{L}_i = &\quad P_{n-1}...P_{i+1}(I - \mathbf{g}_i\mathbf{e}_i^T)P_{i+1}...P_{n-1} \\ = &\quad I - (P_{n-1}...P_{i+1}\mathbf{g}_i)(\mathbf{e}_i^TP_{i+1}...P_{n-1}) \equiv I - \hat{\mathbf{g}}_i\mathbf{e}_i^T. \end{aligned}$$

Il vettore $\hat{\mathbf{g}}_i$ ha la stessa struttura del vettore elementare di Gauss prima definito ma con le ultime $n-i$ componenti permutate tra loro. Ne consegue che la struttura della matrice $\hat{L}_i$ è analoga a quella della matrice elementare di Gauss prima definita; si può quindi concludere che la matrice $\hat{L}_{n-1}...\hat{L}_1 \equiv L^{-1}$ è triangolare inferiore a matrice unitaria. Abbiamo quindi ottenuto una fattorizzazione con pivotin parziale:

$$PA = LU$$

Teorema

Se A è una matrice nonsingolare, allora esiste una matrice di permutazione P tale che PA è fattorizzabile LU.

Osservazione:

Questo metodo permette di estendere l'utilizzo della fattorizzazione LU, infatti si ha che:

$$A\mathbf{x} = \mathbf{b} \Leftrightarrow PA\mathbf{x} = P\mathbf{b} \Leftrightarrow (L\mathbf{y} = P\mathbf{b} \quad \wedge \quad U\mathbf{x} = \mathbf{y}).$$

e la matrice di permutazione P serve soltanto per permutare il vettore dei termini noti $\mathbf{b}$, operazione che precede direttamente la risoluzione dei sistemi triangiolari sopra descritti nell'ordine specificato.

Implementazione

Questo algoritmo implementa la fattorizzazione LU con pivoting appena descritta:

```
%Fattorizzazione con pivoting parziale
%
% [A,p] = fattPivotlib(A)
%
%Questo metodo prende in input:
% A:  una matrice
%E restituisce:
% A: la matrice contenente la fattorizzazione
% p: il vettore di permutazione
% t: il tempo di computazione

function [A,p] = fattPivotlib(A)

n = length(A);

p = [1:n];
for i = 1:n-1
    [mi,ki] = max(abs(A(i:n,i))); %mi elemento massimo,
  %ki indice riga relativo sottomatrice
    if mi==0, error('Matrice singolare'), end
    ki = ki +i-1;
    if ki > i
        A([i ki],:) = A([ki i],:);
        p([i ki]) = p([ki i]);
    end
    A(i+1:n,i) = A(i+1:n,i)/A(i,i);
    A(i+1:n,i+1:n)=A(i+1:n,i+1:n)-A(i+1:n,i)*A(i,i+1:n);
end
```

3.7 Condizionamento del problema

Studieremo adesso in che modo delle perturbazioni sui dati di un sistema lineare del tipo $A\mathbf{x} = \mathbf{b}$ si ripercuotono sulla sua soluzione.

Dobbiamo prima dare delle definizioni:

Norma

Sia $\mathbf{x} \in \mathbb{R}^n$, la norma di un vettore è una funzione definita su uno spazio vettoriale V e codominio in $\mathbb{R}$ tale che:

1. $\forall \mathbf{x} \in V: \quad \|\mathbf{x}\| \geq 0, \quad \|\mathbf{x}\| = 0 \Leftrightarrow \mathbf{x} = 0.$
2. $\forall \mathbf{x} \in V$ e $\forall \alpha \in \mathbb{R}: \quad \|\alpha\mathbf{x}\| = |\alpha| \cdot \|\mathbf{x}\|.$
3. $\forall \mathbf{x},\mathbf{y} \in V, \quad \|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$

Norma "p"

Sia $V \in \mathbb{R}^n$. Definiamo:

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |\mathbf{x}_i|^p \right)^{\frac{1}{p}}, \quad p > 0.$$

Piu in dettaglio:

$$p = \begin{cases} = 1 & \text{Norma 1} & \|\mathbf{x}\|_1 = \sum_{i=1}^n |\mathbf{x}_i| \\ = 2 & \text{Norma euclidea} & \|\mathbf{x}\|_2 = \left(\sum_{i=1}^n |\mathbf{x}_i|^2\right)^{\frac{1}{2}} \\ = \infty & \text{Norma infinito} & \lim_{p\to\infty} \|\mathbf{x}\|_p \Rightarrow max(|\mathbf{x}_i|), \quad i = 1, ..., n. \end{cases}$$

Verifichiamo adesso che le norme appena descritte verificano le tre proprietà sopra elencate.
Per quanto riguarda la prima proprietà si ha che:

- $\|\mathbf{x}\|_1 \geq 0$: $\|\mathbf{x}\|_1 = \sum_{i=1}^n |\mathbf{x}_i|$ ovvero sommatoria di tutti numeri positivi. Deve quindi essere positiva ed uguale a zero solo se $\mathbf{x}_i = 0, \quad i = 0, ..., n$.
- $\|\mathbf{x}\|_2 \geq 0$: $\|\mathbf{x}\|_2 = \left(\sum_{i=1}^n |\mathbf{x}_i|^2\right)^{\frac{1}{2}}$ anche in questo caso abbiamo un sommatoria di soli valori positivi e vale quanto appena detto per la norma 1.
- $\|\mathbf{x}\|_\infty \geq 0$: $\|\mathbf{x}\|_\infty = max(|\mathbf{x}_i|)$, il massimo tra valori assoluti non può che essere un valore positivo e uguale a zero solo se tutti gli $\mathbf{x}_i$ sono nulli.

Per la seconda si ha che:

- $\|\alpha\mathbf{x}\|_1 = |\alpha|\|\mathbf{x}\|_1$: $\|\alpha\mathbf{x}\|_1 = \sum_{i=1}^n |\alpha\mathbf{x}_i| = |\alpha| \sum_{i=1}^n |\mathbf{x}_i|$.
- $\|\alpha\mathbf{x}\|_2 = |\alpha|\|\mathbf{x}\|_2$: $\|\alpha\mathbf{x}\|_2 = (\sum_{i=1}^n |\alpha\mathbf{x}_i|^2)^{\frac{1}{2}} = (|\alpha|^2 \sum_{i=1}^n |\mathbf{x}_i|^2)^{\frac{1}{2}} = |\alpha|(\sum_{i=1}^n |\mathbf{x}_i|^2)^{\frac{1}{2}}$.
- $\|\alpha\mathbf{x}\|_\infty = |\alpha|\|\mathbf{x}\|_\infty$: $\|\alpha\mathbf{x}\|_\infty = max(|\alpha\mathbf{x}_i|) = |\alpha| max(|\mathbf{x}_i|)$

E infine per la terza:

- $\|\mathbf{x}+\mathbf{y}\|_1 \leq \|\mathbf{x}\|_1 + \|\mathbf{y}\|_1$: $\sum_{i=1}^n |\mathbf{x}_i + \mathbf{y}_i| \leq \sum_{i=1}^n |\mathbf{x}_i| + \sum_{i=1}^n |\mathbf{y}_i|$ in quanto vale la disuguaglianza triangolare per i valori assoluti.
- $\|\mathbf{x}+\mathbf{y}\|_2 \leq \|\mathbf{x}\|_2 + \|\mathbf{y}\|_2$: $(\sum_{i=1}^n |\mathbf{x}_i + \mathbf{y}_i|)^{\frac{1}{2}} \leq (\sum_{i=1}^n |\mathbf{x}_i| + \sum_{i=1}^n |\mathbf{y}_i|)^{\frac{1}{2}}$ in quanto vale la disuguaglianza triangolare per i valori assoluti.
- $\|\mathbf{x}+\mathbf{y}\|_\infty \leq \|\mathbf{x}\|_\infty + \|\mathbf{y}\|_\infty$: $\|\mathbf{x}+\mathbf{y}\|_\infty = max(|\mathbf{x}_i + \mathbf{y}_i|) \leq max(|\mathbf{x}_i|) + max(|\mathbf{y}_i|)$ ancora per la disuaguaglianza triangolare sui valori assoluti.

Notiamo inoltre che le norme sono tra loro equivalenti, vale infatti che:

$$\forall p, q, \quad \exists \quad \gamma_1, \gamma_2 \quad | \quad \gamma_1, \gamma_2 > 0, \quad \gamma_1 < \gamma_2, \quad \gamma_1\|\mathbf{x}\|_p \leq \|\mathbf{x}\|_q \leq \gamma_2\|\mathbf{x}\|_p$$

Norma indotta su matrici

Sia $V = \mathbb{R}^{m\times n}$ e $A \in \mathbb{R}^{m\times n}$. Definiamo la norma "p" su una matrice indotta dalla corrispondente norma "p" su vettore come:

$$\|A\|_p = \max_{\|\mathbf{x}\|_p} \|A\mathbf{x}\|_p$$

dove $\|A\|$ e $\|\mathbf{x}\|$ possono essere su spazi diversi se A è rettangolare.

Andiamo a studiare il sistema lineare:

$$(A + \Delta A)(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{b} + \Delta\mathbf{b}$$

dove le perturbazioni ΔA e $\Delta\mathbf{b}$ determinano la perturbazione $\Delta\mathbf{x}$ sulla soluzione. Consideriamo, per semplicità, il caso in cui i dati dipendano da un parametro scalare di perturbazione di dimensioni prossime a zero:

$$A(\varepsilon) = A + \varepsilon F, \quad F \in \mathbb{R}^{n\times n} \Longrightarrow \Delta A = \varepsilon F,$$

$$\mathbf{b}(\varepsilon) = \mathbf{b} + \varepsilon\mathbf{f}, f \in \mathbb{R}^n \Longrightarrow \Delta\mathbf{b} = \varepsilon\mathbf{f}.$$

Da cui consegue:

$$A(\varepsilon)\mathbf{x}(\varepsilon) = \mathbf{b}(\varepsilon),$$

Si osservi che:

$$A(0) = A, \quad \mathbf{b}(0) = \mathbf{b}, \quad \Longrightarrow \mathbf{x}(0) = \mathbf{x}.$$

Sviluppando in serie di Taylor in $\varepsilon = 0$, si ottiene:

$$\mathbf{x}(\varepsilon) = \mathbf{x} + \varepsilon\mathbf{x}'(0) + O(\varepsilon^2) \approx \mathbf{x} + \varepsilon\mathbf{x}'(0)$$

cioè

$$\Delta\mathbf{x} \equiv \mathbf{x}(\varepsilon) - \mathbf{x} \approx \varepsilon\mathbf{x}'(0)$$

Si può inoltre ricavare che:

$$A'(0)\mathbf{x} + A\mathbf{x}'(0) = \mathbf{b}'(0).$$

Quindi:

$$\frac{d}{d\varepsilon}A(\varepsilon)\mathbf{x}(\varepsilon) = \frac{d}{d\varepsilon}\mathbf{b}(\varepsilon) \equiv \mathbf{f}$$

$$= A'(\varepsilon)\mathbf{x}(\varepsilon) + A(\varepsilon)\mathbf{x}'(\varepsilon) = \Big| \ F\mathbf{x}(\varepsilon) + A(\varepsilon)\mathbf{x}'(\varepsilon) \ \Big|_{\varepsilon=0} = F\mathbf{x} + A(0) + \mathbf{x}'(0) = \mathbf{f}$$

che permette di ottenere:

$$\mathbf{x}'(0) = A^{-1}(\mathbf{f} - F\mathbf{x}). \Rightarrow \varepsilon\mathbf{x}'(0) = A^{-1}(\varepsilon\mathbf{f} - \varepsilon F\mathbf{x}) \Rightarrow \Delta\mathbf{x} \approx A^{-1}(\Delta\mathbf{b} - \Delta A\mathbf{x}).$$

Passando alle norme:

$$\|\Delta\mathbf{x}\| = \|A^{-1}(\Delta\mathbf{b} - \Delta A\mathbf{x})\|$$

Ma per la disugualianza triangolare si può affermare che:

$$\|\Delta\mathbf{x}\| \leq \|A^{-1}\| \cdot \|\Delta\mathbf{b} - \Delta A\mathbf{x}\| \leq \|A^{-1}\|(\|\Delta\mathbf{b}\| + \|\Delta A\| \cdot \|\mathbf{x}\|)$$

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \|A^{-1}\| \left(\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{x}\|} + \|\Delta A\|\right) = \|A\| \cdot \|A^{-1}\| \left(\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{x}\| \cdot \|A\|} + \frac{\|\Delta A\|}{\|A\|}\right)$$

ma tenendo condo che:

$$\mathbf{b} = A\mathbf{x}, \quad \|\mathbf{b} \leq \|A\| \cdot \|\mathbf{x}\|;$$

si può concludere che:

$$\|A\| \cdot \|A^{-1}\| \left(\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} + \frac{\|\Delta A\|}{\|A\|}\right)$$

e quindi:

$$\underbrace{\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|}}_{\text{errore sui dati in uscita}} \leq \|A\| \cdot \|A^{-1}\| \left(\underbrace{\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} + \frac{\|\Delta A\|}{\|A\|}}_{\text{errore sui dati in ingresso}}\right)$$

Abbiamo ottenuto una equazione che misura una sorta di errore relativo sul risultato, ed in particolare abbiamo che $k = \|A\| \cdot \|A^{-1}\|$ è il numero di condizionamento del problema. É importante notare che $k(A) = \|A\| \cdot \|A^{-1}\| \geq \|A \cdot A^{-1}\| = \|I\| = 1$.

3.8 Sistemi lineari sovradeterminati

Poniamo adesso il caso di voler risolvere un sistema di equazioni lineari in cui ci siano più equazioni che incognite ma in cui la matrice dei coefficienti abbia rango massimo. Formalmente siamo quindi nel caso:

$$\begin{aligned} &A \in \mathbb{R}^{m\times n}, \\ &\mathbf{x} \in \mathbb{R}^n, \quad \mathbf{b} \in \mathbb{R}^m, \\ &n = rank(A). \end{aligned}$$

In particolare possiamo considerare la matrice A in questo modo:

$$\mathbb{R}^{m\times n} \ni A = (C_1 C_2 ... C_n), \quad C_j \in R^m$$

si ha che:

$$span(A) = \left\{ \mathbf{y} \in \mathbb{R}^m \,\middle|\, \mathbf{y} = \sum_{j=1}^{n} \alpha_j C_j \right\}$$

dove

$$x = \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix} \Rightarrow \mathbf{y} = A\mathbf{x}$$

ed

$$n = rank(A) = dim(span(A))$$

Possiamo quindi affermare che:

$$A\mathbf{x} = \mathbf{b} \text{ ha soluzione } \iff \mathbf{b} \in span(A).$$

Definizione: spazio nullo

$$null(A) = \{\mathbf{x} \in \mathbb{R}^n \,|\, A\mathbf{x} = \mathbf{0}\}$$

Se la matrice ha rango massimo $null(A)$ contiene un solo vettore; generalmente si ha:

$$dim(null(A)) = n - rank(A)$$

se il rango della matrice non è massimo.
In particolare avremo che:

$$n - rank(A) = \begin{cases} = 0, rank(A) = n & \iff null(A) = \{\mathbf{0}\} \\ > 0, < n & \implies \exists \inf^k \text{ vettori in } null(A). \end{cases}$$

Dobbiamo notare che:

$$\forall \mathbf{v} \in null(A), \quad A(\mathbf{x} + \mathbf{v}) = A\mathbf{x} + \underbrace{A\mathbf{v}}_{=0} = \mathbf{b}$$

ovvero se $rank(A)$ non è massimo ci sono infinite soluzioni.

Problema dell'esistenza della soluzione

Sia $\mathbf{b} \in \mathbb{R}^m$ un vettore e $span(A) = n < m$ il rango della matrice. Allora si ha che $span(A) \subsetneqq \mathbb{R}^m$. Questo significa che l'insieme differenza $(\mathbb{R}^m - span(A))$ è molto grande e la situazione $\mathbf{b} \in span(A)$ risulta essere un evento piu "fortunato" che probabile, ovvero in generale non si può assumere che $\mathbf{b}$ sia nello spazio di A.

Definizione: Vettore residuo

Per risolvere il sistema sopra descritto proveremo dunque a trovare il vettore $\mathbf{x}$ che minimizza il vettore:

$$\mathbf{r} = \begin{pmatrix} r_1 \\ \vdots \\ r_m \end{pmatrix} \equiv A\mathbf{x} - \mathbf{b}$$

Ovvero andremo a cercare la suluzione del sistema nel senso dei minimi quadrati:

$$\sum_{i=1}^{m} |\mathbf{r}_i|^2 = \|\mathbf{r}\|_2^2 = \|A\mathbf{x} - \mathbf{b}\|_2^2$$

Teorema

Se A è una matrice in $\mathbb{R}^{m\times n}, m > n, rank(A) = n$ allora esistono $Q \in \mathbb{R}^{m\times m}, Q^TQ = I_m$ e $\hat{R} \in \mathbb{R}^{n\times n}$ triangolare superiore e nonsingolare tali che:

$$A = QR \equiv Q \begin{pmatrix} \hat{R} \\ \mathbf{0} \end{pmatrix}, \quad R \in \mathbb{R}^{m\times n}$$

É importante ricordare la seguente proprietà delle matrici ortogonali:

$$\|Q\mathbf{v}\|_2^2 = (Q\mathbf{v})^TQ\mathbf{v} = \mathbf{v}^TQ^TQ\mathbf{v} = \mathbf{v}^T\mathbf{v} = \|\mathbf{v}\|_2^2$$

ovvero la norma euclidea è invariante per moltiplicazioni per matrici ortogonali a sinistra o a destra.
Vediamo su quali fattori lavorare per minimizzare il vettore residuo:

$$\begin{aligned} &\|A\mathbf{x} - \mathbf{b}\|_2^2 = \|QR\mathbf{x} - \mathbf{b}\|_2^2 = \|Q(R\mathbf{x} - Q^T\mathbf{b})\|_2^2 = \|R\mathbf{x} - \mathbf{g}\|_2^2 \\ &= \left\| \begin{pmatrix} \hat{R} \\ \mathbf{0} \end{pmatrix} \mathbf{x} - \mathbf{g} \right\|_2^2 = \left\| \begin{pmatrix} \hat{R} \\ \mathbf{0} \end{pmatrix} \mathbf{x} - \begin{pmatrix} \mathbf{g}_1 \\ \mathbf{g}_2 \end{pmatrix} \right\|_2^2 = \left\| \begin{pmatrix} \hat{R}\mathbf{x} - \mathbf{g}_1 \\ -\mathbf{g}_2 \end{pmatrix} \right\|_2^2 \\ &= \left\| \hat{R}\mathbf{x} - \mathbf{g}_1 \right\|_2^2 + \|\mathbf{g}_2\|_2^2 \end{aligned}$$

in cui abbiamo definito e quindi utilizzato:

$$\mathbf{g} = Q^T\mathbf{b} = \begin{pmatrix} \mathbf{g}_1 \\ \mathbf{g}_2 \end{pmatrix} \text{ tale che } \mathbf{g}_1 \in \mathbb{R}^n, \quad \mathbf{g}_2 \in \mathbb{R}^{m-n}.$$

Generalmente quindi $\mathbf{b}$ non appartiene allo $span(A)$. Ne deriva quindi che se $\hat{R}\mathbf{x} = \mathbf{g}_1$ allora $\|A\mathbf{x} - \mathbf{b}\|_2^2 = \|\mathbf{g}_2\|_2^2 \equiv \min$. É importante osservare che:

- Il sistema lineare $\hat{R}\mathbf{x} = \mathbf{g}_1$ ammette una unica soluzione in quanto $\hat{R}$ è non singolare, e questa è quindi la soluzione ai minimi quadrati di $A\mathbf{x} = \mathbf{b}$;
- La matrice $\hat{R}$ è triangolare superiore e il sistema è quindi facilmente risolvibile;
- Il fattore Q della fattorizzazione non è esplicitamente richiesto, ci è sufficiente effettuare il prodotto $Q^T\mathbf{b}$ per ottenere il vettore $\mathbf{g}$.

3.8.1 Esistenza della fattorizzazione QR

Matrice elementare di Householder

Dato il vettore:

$$\mathbf{z} = (z_1, ..., z_m)^T \in \mathbb{R}^m, \quad \mathbf{z} \neq 0$$

vogliamo determinare una matrice ortogonale H tale che:

$$H\mathbf{z} = \alpha \mathbf{e}_1, \quad \alpha \in \mathbb{R}$$

Si ha che:

$$\|\mathbf{z}\|_2^2 = \mathbf{z}^T\mathbf{z} = \mathbf{z}^T H^T H \mathbf{z} = \alpha^2 \mathbf{e}_1^T \mathbf{e}_1 = \alpha^2,$$

ovvero possiamo definire α come:

$$\alpha = \pm \|\mathbf{z}\|_2.$$

Consideriamo quindi una matrice della seguente forma:

$$H = I - \frac{2}{\underbrace{\mathbf{v}^T\mathbf{v}}_{=\|\mathbf{v}\|_2^2}} \mathbf{v}\mathbf{v}^T, \quad \mathbf{v} \neq 0,$$

dove $\mathbf{v} \in \mathbb{R}^m$ sarà diverso dal vettore nullo se anche $\mathbf{z}$ lo è. La matrice H così costruita è simmetrica per definizione, ed è anche ortogonale:

$$\begin{aligned} H^T H &= H^2 = \left(I - \frac{2}{\mathbf{v}^T\mathbf{v}}\mathbf{v}\mathbf{v}^T\right)\left(I - \frac{2}{\mathbf{v}^T\mathbf{v}}\mathbf{v}\mathbf{v}^T\right) = I - \frac{4}{\mathbf{v}^T\mathbf{v}}\mathbf{v}\mathbf{v}^T + \frac{4}{(\mathbf{v}^T\mathbf{v})^2}(\mathbf{v}\mathbf{v}^T)^2 \\ &= I - \frac{4}{\mathbf{v}^T\mathbf{v}}\mathbf{v}\mathbf{v}^T + \frac{4}{(\mathbf{v}^T\mathbf{v})(\mathbf{v}^T\mathbf{v})}\left(\mathbf{v}(\mathbf{v}^T\mathbf{v})\mathbf{v}^T\right) = I - \frac{4}{\mathbf{v}^T\mathbf{v}}\mathbf{v}\mathbf{v}^T + \frac{4}{\mathbf{v}^T\mathbf{v}}\mathbf{v}\mathbf{v}^T = I \end{aligned}$$

Scegliamo adesso un vettore

$$v = z - \alpha \mathbf{e}_1$$

e verifichiamo se $H\mathbf{z} = \alpha\mathbf{e}_1, \quad \alpha \in \mathbb{R}$.

$$\begin{aligned} H\mathbf{z} &= \left(I - \frac{2}{\mathbf{v}^T\mathbf{v}}\mathbf{v}\mathbf{v}^T\right)\mathbf{z} = \mathbf{z} - \frac{2}{\mathbf{v}^T\mathbf{v}}\mathbf{v}\mathbf{v}^T\mathbf{z} \\ &\quad \left(\text{tenendo conto che } \mathbf{v}^T\mathbf{z} = (\mathbf{z} - \alpha\mathbf{e}_1)\mathbf{z} = \mathbf{z}^T\mathbf{z} - \alpha\mathbf{e}_1 T\mathbf{z} = \|\mathbf{z}\|_2^2 - \alpha\mathbf{z}_1\right) \\ &= \mathbf{z} - \frac{2}{\mathbf{v}^T\mathbf{v}}\left(\mathbf{z}^T\mathbf{z} - \alpha\mathbf{z}_1\right)\mathbf{v} = \mathbf{z} - \frac{2}{\mathbf{v}^T\mathbf{v}}\left(\mathbf{z}^T\mathbf{z} - \alpha\mathbf{z}_1\right)(\mathbf{z} - \alpha\mathbf{e}_1) \\ &= \left(1 - \underbrace{\frac{2}{\mathbf{v}^T\mathbf{v}}\left(\mathbf{z}^T\mathbf{z} - \alpha\mathbf{z}_1\right)}\right)\mathbf{z} + \alpha\,\underbrace{\frac{2}{\mathbf{v}^T\mathbf{v}}\left(\mathbf{z}^T\mathbf{z} - \alpha\mathbf{z}_1\right)}\,\mathbf{e}_1 \end{aligned}$$

Dall'equazione sopra scritta notiamo che se gli scalari sopra evidenziati fossero di valore pari ad 1 allora otterremmo $H\mathbf{z} = \alpha\mathbf{e}_i$, dimostriamolo:

$$\frac{2}{\mathbf{v}^T\mathbf{v}}(\mathbf{z}^T\mathbf{z} - \alpha\mathbf{z}_1) = \frac{2\|\mathbf{z}\|_2^2 - 2\alpha\mathbf{z}_1}{(\mathbf{z} - \alpha\mathbf{e}_1)^T(\mathbf{z} - \alpha\mathbf{e}_1)} = 1 \quad \Rightarrow \quad 2(\mathbf{z}^T\mathbf{z} - \alpha\mathbf{z}_1) = \mathbf{v}^T\mathbf{v}$$

dove svolgendo $\mathbf{v}^T\mathbf{v}$ otteniamo

$$\mathbf{v}^T\mathbf{v} = (\mathbf{z} - \alpha\mathbf{e}_1)^T(\mathbf{z} - \alpha\mathbf{e}_1) = \mathbf{z}^T\mathbf{z} + \alpha^2 - 2\alpha\underbrace{\mathbf{e}_1^T\mathbf{z}}_{\mathbf{z}_1} \quad \text{considerando} \quad \alpha^2 = \|\mathbf{z}\|^2 = \mathbf{z}^T\mathbf{z}$$

$$\text{allora} \quad \mathbf{v}^T\mathbf{v} = \mathbf{z}^T\mathbf{z} + \mathbf{z}^T\mathbf{z} - 2\alpha\mathbf{z}_1 = 2\mathbf{z}^T\mathbf{z} - 2\alpha\mathbf{z}_1$$

per cui abbiamo verificato

$$2(\mathbf{z}^T\mathbf{z} - \alpha\mathbf{z}_1) = 2\mathbf{z}^T\mathbf{z} - 2\alpha\mathbf{z}_1$$

abbiamo quindi dimostrato che $H\mathbf{z} = \alpha\mathbf{e}_1$.

Discussione del segno di α

$$\mathbf{v} = \mathbf{z} - \alpha\mathbf{e}_1 = \begin{pmatrix} \mathbf{z}_1 - \alpha \\ \mathbf{z}_2 \\ \vdots \\ \mathbf{z}_n \end{pmatrix}$$

tenendo conto del condizionamento della somma algebrica imponiamo:

$$\alpha = -sign(\mathbf{z}_1) \cdot \|\mathbf{z}\|_2$$

Il calcolo di $\mathbf{v}$ avrà pertanto numero di condizionamento pari a uno.
Vediamo adesso come si possa risparmiare una locazione di memoria nella memorizzazione di $\mathbf{v}$:

$$|\mathbf{v}_1| > 0, \quad se \|\mathbf{z}\| > 0$$

possiamo quindi riscrivere $\mathbf{v}$ come:

$$\mathbf{v} = \mathbf{v}_1 \begin{pmatrix} 1 \\ \frac{\mathbf{v}_2}{\mathbf{v}_1} \\ \vdots \\ \frac{\mathbf{v}_m}{\mathbf{v}_1} \end{pmatrix} = \mathbf{v}_1 \cdot \hat{\mathbf{v}} \Rightarrow \hat{\mathbf{v}} = \frac{1}{\mathbf{v}_1}\mathbf{v}$$

Notiamo adesso che:

$$H(\mathbf{v}) = \left(I - \frac{2}{\mathbf{v}^T\mathbf{v}}\mathbf{v}\mathbf{v}^T\right)$$

e quindi

$$H(\hat{\mathbf{v}})\left(I - \frac{2}{\hat{\mathbf{v}}^T\hat{\mathbf{v}}}\hat{\mathbf{v}}\hat{\mathbf{v}}^T\right) = \left(I - \frac{2}{\frac{\mathbf{v}^T\mathbf{v}}{\mathbf{v}_1^2}}\frac{1}{\mathbf{v}_1^2}\mathbf{v}\mathbf{v}^T\right) = \left(I - \frac{2}{\mathbf{v}^T\mathbf{v}}\mathbf{v}\mathbf{v}^T\right) = H(\mathbf{v})$$

ovvero si può sostituire il vettore $\mathbf{v}$ con un qualunque altro vettore multiplo scalare di esso e la matrice di Householder non cambia. Possiamo quindi usare il vettore normalizzato e risparmiare una locazione di memoria.

Esistenza della fattorizzazione QR

Dimostriamo adesso in maniera costruttiva l'esistenza della fattorizzazione QR:
Sia:

$$A = \begin{pmatrix} a_{11}^{(0)} & \cdots & a_{1n}^{(0)} \\ \vdots & & \vdots \\ a_{m1}^{(0)} & \cdots & a_{mn}^{(0)} \end{pmatrix} \equiv A^{(0)}.$$

dove l'indice superiore indica il passo più recente in cui è stato modoficato l'elemento corrispondente.
Considerando la prima colonna della matrice $A^{(0)}$ possiamo definire la matrice di Householder $H_1 \in \mathbb{R}^{m\times m}$ tale che:

$$H_1 \begin{pmatrix} a_{11}^{(0)} \\ \cdots \\ a_{m1}^{(0)} \end{pmatrix} \equiv \begin{pmatrix} a_{11}^{(1)} \\ 0 \\ \cdots \\ 0 \end{pmatrix}$$

e se A ha rango massimo si deve necessariamente avere che $a_{11}^{(1)} \neq 0$, altrimenti la prima colonne della matrice risulterebbe nulla. Otterremo quindi:

$$H_1 A^{(0)} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{21}^{(1)} & \cdots & a_{2n}^{(1)} \\ \vdots & \vdots & & \vdots \\ 0 & a_{m2}^{(1)} & \cdots & a_{mn}^{(1)} \end{pmatrix} \equiv A^{(1)}$$

Possiamo adesso iterare il passo fatto in precedenza sulla seconda colonna della matrice $A^{(1)}$ a partire dalla seconda componente e definire quindi una matrice di Householder in questo modo:

$$H_2 \equiv \left(\begin{array}{c|c} 1 & \\ \hline & H^{(2)} \end{array}\right) \in \mathbb{R}^{m\times m}, \quad H^{(2)} \begin{pmatrix} a_{22}^{(1)} \\ \vdots \\ a_{m2}^{(1)} \end{pmatrix} \equiv \begin{pmatrix} a_{22}^{(1)} \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

La seconda matrice di Householder così ottenuta è quindi la composizione della matrice identità con una sottomatrice di Householder di dimensione $m-1$. Notiamo che essa è ancora ortogonale e vale:

$$H_2 A^{(1)} = H_2 H_1 A = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} \\ 0 & 0 & a_{33}^{(2)} & \cdots & a_{3n}^{(2)} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & a_{m3}^{(2)} & \cdots & a_{mn}^{(2)} \end{pmatrix} \equiv A^{(2)}$$

Anche a questo passo si dovrà avere che $a_{22}^{(2)} \neq 0$ se la matrice A ha rango massimo. Generalizzando al passo i-esimo si avrà quindi:

$$A^{(i)} \equiv H_i H_{i-1} \cdots H_1 A = \begin{pmatrix} a_{11}^{(1)} & \cdots & \cdots & \cdots & \cdots & a_{1n}^{(1)} \\ 0 & \ddots & & & & \vdots \\ \vdots & \ddots & a_{ii}^{(i)} & \cdots & \cdots & a_{in}^{(i)} \\ \vdots & & 0 & a_{i+1,i+1}^{(i)} & \cdots & a_{i+1,n}^{(i)} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a_{m,i+1}^{(i)} & \cdots & a_{mn}^{(i)} \end{pmatrix}$$

con $a_{jj}^{(j)} \neq 0, \quad j = 1, ..., i.$
Possiamo quindi definire la matrice di Householder al passo $i+1$ come:

$$H_{i+1} = \left(\begin{array}{c|c} I_i & \\ \hline & H^{(i+1)} \end{array}\right) \in \mathbb{R}^{m\times m}, \quad H^{(i+1)} \begin{pmatrix} a_{i+1,i+1}^{(i)} \\ \vdots \\ a_{m,i+1}^{(i)} \end{pmatrix} \equiv \begin{pmatrix} a_{i+1,i+1}^{(i)} \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

ottenendo così ancora una matrice ortogonale che soddisfa:

$$H_{i+1} A^{(i)} = H_{i+1} H_i \cdots H_1 A \equiv A^{(i+1)}.$$

con $a_{i+1,i+1}^{(i+1)} \neq 0$. Procedento in questo modo dopo n passi si arriva alla matrice:

$$A^{(n)} \equiv H_n \cdots H_1 A = \begin{pmatrix} a_{11}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & \ddots & \vdots \\ \vdots & \ddots & a_{nn}^{(n)} \\ \vdots & & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{pmatrix} \equiv R$$

con $a_{ii}^{(i)} \neq 0, \quad i = 1, ..., n$. Se poniamo quindi $Q^T = H_n \cdots H_1$, abbiamo ottenuto la fattorizzazione $A = QR$.

Implementazione

```
%Fattorizzazione QR con il metodo di Householder
%
% [A] = fattQRhouseholder(A)
%
%Questo metodo prende in input:
% A: matrice rettangolare mxn con m>>n rank(A)=n
%E restituisce:
% A: matrice mxn contenente la fattorizzazione
% t: il tempo di esecuzione

function A = fattQRhouseholder(A)

[m,n] = size(A);
for i = 1:n
    alfa = norm(A(i:m,i));
    if alfa == 0, error('La matrice A non ha rango massimo'), end
    if A(i,i) >= 0, alfa = -alfa; end
    v1 = A(i,i)-alfa;
    A(i,i) = alfa;
    A(i+1:m,i) = A(1+i:m,i)/v1;
    beta = -v1/alfa;

    for c =  i+1:n
        A(i:m,c) = A(i:m,c)-(beta*[1;A(i+1:m,i)])*([1 A(i+1:m,i)']*A(i:m,c));
    end
    %A(i:m,i+1:n) = A(i:m,i+1:n)-(beta*[1;A(i+1:m,i)])...
    % *([1 A(i+1:m,i)']*A(i:m,i+1:n));
end
```

	Flops	Memoria
LU	$O(\frac{2}{3}n^3)$	n^2
LDL	$O(\frac{1}{3}n^3)$	$\frac{n^2}{2} + O(n)$
PLU	$\frac{2}{3}n^3$	$n^2 + O(n)$
QR	$O(\frac{2}{3}n^2(3m - n))$	$O(m * n)$

Tabella 3.1: Tabella comparativa dei costi di fattorizzazione

Capitolo 4

Approssimazione di funzioni

Il problema che ci poniamo in questo capitolo è quello di determinare una conveniente approssimazione di una generica funzione

$$f : [a, b] \subset \mathbb{R} \to \mathbb{R}$$

Questo può rendersi necessario per diversi motivi:

- la forma funzionale di f potrebbe essere troppo complessa;
- la forma funzionale di f potrebbe essere non nota, e in questo caso si ipotizza di essere a conoscenza dei valori assunti su un insieme di ascisse tra loro distinte: $a \leq x_0 < x_1 < ... < x_n \leq b$.

4.1 Interpolazione polinomiale

Avendo a disposizione i seguenti dati:

$$(x_i, f_i), \quad i = 0, ..., n, \quad f_i \equiv f(x_i),$$

ricerchamo un polinomio interpolante la $f(x)$ sulle ascisse $x_i \in [a, b], \quad x_i < x_j$ se $i < j$,$x_i \neq x_j$ se $i \neq j$, tale che:

$$p(x) \in \Pi_n, \quad p(x_i) = f_i, \quad i = 0, .., n.$$

Teorema

$$\exists!\ p_n \in \Pi_n \text{ tale che } p_n(x_i) = f_i, \quad i = 0, .., n.$$

Dimostrazione:
Un generico polinomio di grado n sarà nella forma:

$$p(x) = \sum_{k=0}^{n} a_k x^k$$

che descrive il sistema di equazioni lineari:

$$V\mathbf{a} = \mathbf{f}, \quad \longrightarrow \quad \begin{pmatrix} x_0^0 & x_0^1 & \cdots & x_0^n \\ x_1^0 & x_1^1 & \cdots & x_1^n \\ \vdots & \vdots & & \vdots \\ x_n^0 & x_n^1 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{pmatrix}$$

in cui V è una matrice di Vandermonde trasposta univocamente definita dalle ascisse x_i. Il determinante di tale matrice vale:

$$det(V) = \prod_{i>j}(x_i - x_j)$$

In quanto per ipotesi le ascisse sono tra loro distinte, abbiamo che V è nonsingolare e quindi esiste ed è unica la soluzione del sistema lineare, ovvero esiste ed è unico il polinomio suddetto. Non utilizzeremo però questa matrice per determinare in polinomio, in quanto tra le altre cose sappiamo anche che essa diviene molto mal condizionata al crescere di n.

4.2 Forma di Lagrange e forma di Newton

Per ottenere un problema discreto con proprietà piu favorevoli rispetto a quello $V\mathbf{a} = \mathbf{f}$ sopra descritto, si può pensare di utilizzare basi diverse per Π_n.
Ad esempio la base di Lagrange:

$$L_{kn}(x) = \prod_{j=0, j\neq k}^{n} \frac{x - x_j}{x_k - x_j}, \quad k = 0, 1, ..., n.$$

I polinomi appena descritti sono tutti ben definiti in quanto, per ipotesi, $x_i \neq x_j$ con $i \neq j$.

Lemma

Dati i polinomi di Lagrange appena descritti definiti sulle ascisse $a \leq x_0 < x_1 < ... < x_n \leq b$:

$$L_{kn}(x_i) = \begin{cases} 1, & \text{se } k = i, \\ 0, & \text{se } k \neq i, \end{cases}$$

inoltre:

- essi hanno grado esatto n e il coefficiente principale di $L_{kn}(x)$ è:

$$c_{kn} = \frac{1}{\prod_{j=0, j\neq k}^{n}(x_k - x_j)}, \quad k = 0, ..., n$$

- questi sono linearmente indiendenti tra loro, ovvero costituiscono una base per Π_n.

Teorema (forma di Lagrange)

Il polinomio

$$p(x) = \sum_{k=0}^{n} f_k L_{kn}(x)$$

appartiene a Π_n e soddisfa i vincoli di interpolazione $p(x_i) = f_i, \quad i = 0, 1, ..., n.$
Dimostrazione:
$p(x) \in \Pi_n$ in quanto per definizione produttoria di n termini di grado 1; inoltre si ha che:

$$p(x) = \sum_{k=0}^{n} f_k L_{kn}(x_i) = \underbrace{\sum_{k=0, k\neq i}^{n} f_k L_{kn}(x_i)}_{=0 \text{ per def.}} + f_i L_{in}(x_i) = f_i.$$

La base di Lagrange genera una forma del polinomio interpolante in cui i coefficienti sono facili da calcolare; questa rappresentazione però si presta male alla computazione in quanto non consente di generare il polinomio in maniera incrementale ma, dovendo calcolare e poi sommare k polinomi di grado n, ha un costo computazionale molto elevato.

Base di Newton

Indichiamo con $p_r(x)$ il polinomio in Π_r che interseca la funzione da approssimare sulle ascisse $x_0, x_1, ..., x_r$ che supponiamo noto, possiamo definire il successivo $p_{r+1}(x)$ a partire da questo, e questo può essere fatto in modo semplice utilizzando la base di Newton così definita:

$$\begin{aligned} &\omega_0(x) \equiv 1 \\ &\omega_{k+1} = (x - x_k)\omega_k(x), \quad k = 0, 1, 2, ... \end{aligned}$$

che gode delle seguenti proprietà:

1) $\omega_k(x) \in \Pi'_k$, ovvero è un polinomio monico di grado k,
2) $\omega_{k+1}(x) = \prod_{j=0}^{k}(x - x_j)$,
3) $\omega_{k+1}(x_j) = 0$, per $j \leq k$,
4) $\omega_0(x), ..., \omega_k(x)$ costituiscono una base per Π_k.

Teorema

La famiglia di polinomi interpolanti $p_r(x)_{r=0}^{n}$ tali che:

- $p_r(x) = \Pi_r$,
- $p_r(x_i) = f_i, \quad i = 0, ..., r$,

si genera ricorsivamente in questo modo:

$$\begin{aligned} &p_0(x) = f_0\omega_0(x) \\ &p_r(x) = p_{r-1}(x) + f[x_0, x_1, ..., x_r]\omega_r(x), \quad r = 1, .., n. \end{aligned} \tag{4.1}$$

dove:

$$f[x_0, x_1, ..., x_r] = \sum_{k=0}^{r} \frac{f_k}{\prod_{j=0, j\neq k}^{r}(x_k - x_j)}. \tag{4.2}$$

e la quantità $f[x_0, x_1, ..., x_r]$ è detta differenza divisa di ordine r della funzione $f(x)$ sulle ascisse $x_0, x_1, ..., x_r$.
Dimostrazione:
Ragionando per induzione, la tesi è banalmente vera per $r = 0$. Supponendola quindi vera per $r - 1$ vogliamo dimostrare che è vera anche per r:

$$p_r(x) \in \Pi_r, \text{ infatti } p_{r-1}(x_i) \in \Pi_{r-1} \text{ e } \omega_r(x) \in \Pi'_r.$$

Inoltre per $i < r$ si ha:

$$p_r(x_i) = p_{r-1}(x_i) + f[x_0, x_1, ..., x_r]\omega_r(x_i) = p_{r-1}(x_i) = f_i$$

in quanto $\omega_r(x_i) = 0$ e per l'ipotesi di induzione. Se $i = r$:

$$p_r(x_r) = p_{r-1}(x_r) + f[x_0, x_1, ..., x_r]\omega_r(x_r) = f_r$$

e quindi

$$f[x_0, x_1, ..., x_r] = \frac{f_r - p_{r-1}(x_r)}{\omega_r(x_r)}.$$

L'espressione è ben definita in quanto, essendo le ascisse distinte, $\omega_r(x_r) \neq 0$. Verifichiamo adesso che l'espressione appena ottenuta è formalmente uguale a quella (4.2): esprimendo il polinomio $p_r(x)$ secondo la base di Lagrange:

$$p_r(x) = \sum_{k=0}^{r} f_k L_{kr}(x)$$

Per l'unicità del polinomio interpolante questi polinomi devono conicidere con quelli in (4.1). I loro coefficienti principali devono quindi a loro volta conicidere.
Considerando la natura iterativa della forma di Newton si ottiene l'espressione:

$$p(x) \equiv p_n(x) = \sum_{r=0}^{n} f[x_0, ..., x_r]\omega_r(x).$$

Osservazione:

Il denominatore di $L_{kn}(x)$ è dato da $\omega'_{n+1}(x_k)$, mentre quello di $f[x_0, x_1, ..., x_r]$ è $\omega'_{r+1}(x_k)$
Le differenze divise di $f(x)$ sulle ascisse $a \leq x_0 < x_1 < ... < x_n \leq b$ definisco i coefficienti della rappresentazione del polinomio interpolante $p(x_i)$ rispetto alla base di Newton.

Teorema

Le differenze divise di $f(x)$ sulle ascisse $a \leq x_0 < x_1 < ... < x_n \leq b$ godono delle seguenti proprietà:

1. se $\alpha, \beta \in \mathbb{R}$ e $f(x), g(x)$ sono funzioni di una variabile reale,

$$(\alpha \cdot f + \beta \cdot g)[x_0, ..., x_r])\alpha \cdot f[x_0, ..., x_r] + \beta \cdot g[x_0, ..., x_r];$$

2. per ogni $i_0, i_1, ..., i_r$ permutazione di $0, 1, ..., r$,

$$f[x_{i0}, ..., x_{ir}] = f[x_0, ..., x_r];$$

3. sia $f(x) = \sum_{k=0}^{k} a_i x^i \in \Pi_k$, allora

$$f[x_0, x_1, ..., x_r] = \begin{cases} a_k, & \text{se } r = k, \\ 0, & \text{se } r > k; \end{cases}$$

4. se $f(x) \in \mathcal{C}^{(r)}$, allora:

$$f[x_0, x_1, ..., x_r] = \frac{f^{(r)}(\varepsilon)}{r!}, \quad \varepsilon \in [\min_i x_i, \max_i x_i];$$

5.

$$f[x_0, x_1, ..., x_{r-1}, x_r] = \frac{f[x_1, ..., x_r] - f[x_0, ..., x_{r-1}]}{x_r - x_0}$$

Osservazione

Dal punto quattro consegue che se le ascisse coincidono ($x_0 = ... = x_n$), il polinomio $p_n(x) = \sum_{r=0}^{n} f[x_0, ..., x_r]\omega_r(x)$ diventa il polinomio di Taylor di $f(x)$ di punto iniziale x_0.
La proprietà descritta nel punto cinque è invece molto interessante dal punto di vista algoritmico: spiega infatti come sia possibile costruire una differenza divisa di ordine r a partire da due differenze divise di ordine $r-1$ che differiscono solo per una delle ascisse. Tenendo conto che $f[x_i] = f_i, \quad i = 0, 1, ..., n$, siamo in grado di generare la seguente tabella triangolare, in cui la diagonale principale contiene i coefficienti del polinomio interpolante nella forma di Newton:

	0	1	2	$\cdots$	$n-1$	n
x_0	$\boxed{f[x_0]}$					
x_1	$f[x_1]$	$\boxed{f[x_0, x_1]}$				
$\vdots$	$\vdots$	$f[x_1, x_2]$	$\boxed{f[x_0, x_1, x_2]}$			
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$		
x_{n-1}	$f[x_{n-1}]$	$f[x_{n-2}, x_{n-1}]$	$\vdots$		$\boxed{f[x_0, ..., x_{n-1}]}$	
x_n	$f[x_n]$	$f[x_{n-1}, x_n]$	$f[x_{n-2}, x_{n-1}, x_n]$	$\cdots$	$f[x_1, ..., x_n]$	$\boxed{f[x_0, ..., x_n]}$

Notiamo che le colonne della tabella vanno calcolate da sinistra verso destra; possiamo però calcolare ciascuna colonna dal basso verso l'alto: in questo modo gli elementi adiacenti a sinistra non sono più necessari nel seguito e si posso quindi riscrivere con il nuovo elemento appena calcolato.

Implementazione

Questo codice implementa il calcolo delle differenze divise appena descritto:

```
%Calcolo delle differenze divise per il polinomio di Newton
%
% f = NewtonDiffDiv(x,f)
%
%Questo metodo prende in input:
% x: vettore delle ascisse
% f: vettore dei valori della funzione
%
%e restituisce:
% f: vettore contenente le differenze divise f[x0],f[x1,x2]...f[x0...xn]
%
%
function f = NewtonDiffDiv(x,f)

n = length(f);
if n ~= length(x), error('Vettori f ed x non compatibili'), end

%Assumiamo che le ascisse siano tutte distinte.
for i = 1:n-1
    for j = n:-1:i+1
        f(j) = (f(j)-f(j-1))/(x(j)-x(j-i));
    end
```

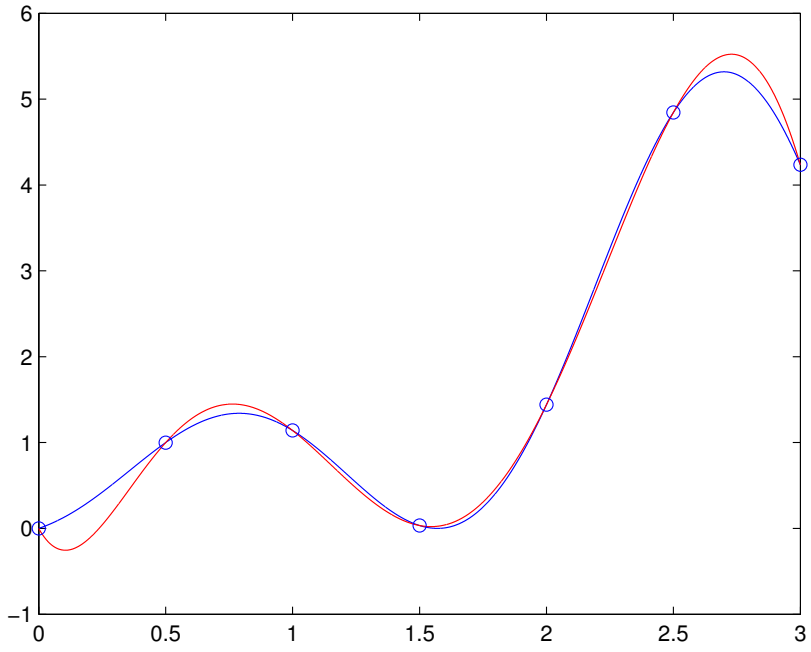

Figura 4.1: Questo grafico mostra l'andamento del polinomio di Newton (in rosso) interpolante la funzione $xsin(3x) + x$ sulle ascisse equidistanti $[0 : 0.5 : 3]$.

```
end
```

4.3 Interpolazione di Hermite

Consideriamo le seguenti ascisse di interpolazione:

$$a \leq x_0 < x_{\frac{1}{2}} < ... < x_n < x_{n+\frac{1}{2}} \leq b = 2n + 2 \text{ ascisse,}$$

$$p(x) \in \Pi_{2n+1} \text{ tali che } p(x_i) = f_i, \quad i = 0, \frac{1}{2}, 1, ..., n + \frac{1}{2}.$$

Se facciamo in modo che:

$$x_{i+\frac{1}{2}} \to x_i, \quad i = 0, 1, ..., m, \text{ e} \quad p(x_i) = f(x_i), p(x_{i+\frac{1}{2}}) = f(x_{i+\frac{1}{2}}),$$

si ha che:

$$p(x_{i+\frac{1}{2}}) - p(x_i) = f(x_{i+\frac{1}{2}}) - f(x_i)$$

e dividendo membro a membro per $x_{i+\frac{1}{2}} - x_i$ si ottiene:

$$p[x_i, x_{i+\frac{1}{2}}] = f[x_i, x_{i+\frac{1}{2}}] \Rightarrow p'(x_i) = f'(x_i)$$

Come conseguenza abbiamo qundi che il polinomio $p(x)$ è ancora definito e soddisfa i vincoli di interpolazione:

$$p(x_i) = f(x_i), \quad p'(x_i) = f'(x_i), \quad i = 0, 1, ..., n.$$

Abbiamo appena definito il polinomio interpolante di Hermite.
Passando al polinomio di Taylor, considerando di avere una sola ascissa:

$$p(x)) = f[x_0] + f[x_0, x_0](x - x_0) \equiv f(x_0) + f'(x_0)(x - x_0)$$

Per generalizzare, se si considerano le ascisse in questo modo:

$$a \leq x_0 = x_0 < x_1 = x_1 < ... < x_n = x_n \leq b,$$

il polinomio di Hermite sarà dato da:

$$\begin{aligned} p(x) \quad = \quad & f[x_0] + f[x_0, x_0](x - x_0) + f[x_0, x_0, x_1](x - x_0)^2 + \\ & f[x_0, x_0, x_1, x_1](x - x_0)^2(x - x_1) + \\ & f[x_0, x_0, x_1, x_1, x_2](x - x_0)^2(x - x_1)^2 + \cdots + \\ & f[x_0, x_0, x_1, x_1, ..., x_m, x_m](x - x_0)^2 \cdots (x - x_{m-1})^2(x - x_m). \end{aligned}$$

Per calcolare i coefficienti ci serviremo dell'algoritmo implementato qui di seguito, che non è altro che una modifica all'algoritmo per il calcolo delle differenze divise. In questo algoritmo avrà in ingresso un vettore **f** contenente i valori:

$$f(x_0), f'(x_0), f(x_1), f'(x_1), ..., f(x_m), f'(x_m),$$

ed un vettore **x** contenente le ascisse.

Implementazione

Questo codice implementa il calcolo delle differenze divise appena descritto:

```
%Calcolo delle differenze divise per il polinomio di Hermite
%
% f = HermiteDiffDiv(x,f)
%
%Questo metodo prende in input:
% x: vettore delle ascisse
% f: vettore dei valori della funzione
%
%e restituisce:
% f: vettore contenente le differenze divise f[x0],f[x0,x0]...f[x0...xn]
%
%

function f = HermiteDiffDiv(x,f)


n = length(f);

for i = n-1:-2:3
    f(i) = (f(i) - f(i-2))/(x(i) - x(i-2));
end


for i = 2:n
    for j = n:-1:i+1
        f(j) = (f(j)-f(j-1))/(x(j)-x(j-i));
    end
end
```

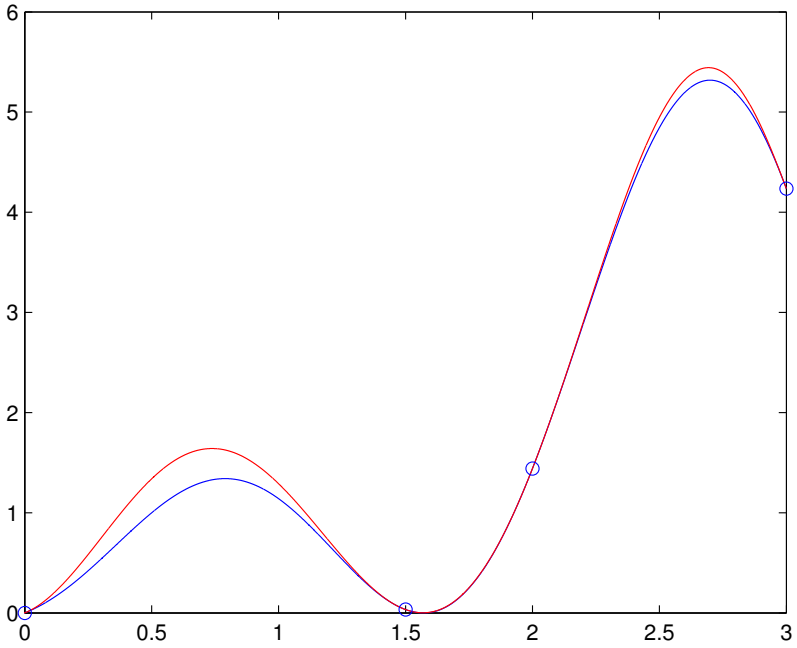

Figura 4.2: Questo grafico mostra l'andamento del polinomio di Hermite (in rosso) interpolante la funzione $xsin(3x)+x$ sulle ascisse $\{0, 1.5, 2, 3\}$; notiamo come già con 4 punti invece dei 7 usati per il polinomio di Newton si abbia un'approssimazione della funzione molto buona.

4.4 Errore nell'interpolazione

Sia $p(x) \in \Pi_n$ il polinomio interpolante tale che, note le coppie (x_i, f_i), $p(x_i) = f_i$ con $i = 0, 1, ..., n$. Possiamo definire l'errore di interpolazione come:

$$e(x) = f(x) - p(x)$$

dove $e(x)$ è l'errore che si commette nell'approssimare $f(x)$ mediante il suo polinomio interpolante $p(x)$. Segue naturalmente che, per definizione, $e(x_i) = 0, \quad i = 0, 1, ..., n$, ma è di nostro interesse misurare questa funzione quando x non appartiene a $x_0, ..., x_n$.

Teorema

Sia

$$p(x) = \sum_{r=0}^{n} f[x_0, ..., x_r]\omega_r(x),$$

tale da soddisfare i vincoli di interpolazione $p(x_i) = f_i$ con $i = 0, 1, ..., n$. Il suo errore di interpolazione è:

$$e(x) = f[x_0, x_1, ..., x_n, x]\omega_{n+1}(x).$$

Dimostrazione:
Si consideri un generico punto $\bar{x}$ per semplicità distinto dalle ascisse di interpolazione $x_0, ..., x_n$, ed il polinomio $\bar{p}(x) \in \Pi_{n+1}$ che soddisfi sia gli stessi vincoli di interpolazione di $p(x)$ che:

$$\bar{p}(\bar{x}) = f(\bar{x})$$

Dal teorema sulla forma di Newton segue:

$$\bar{p}(\bar{x}) = p(\bar{x}) + f[x_0, x_1, ..., x_m, \bar{x}]\omega_{n+1}(\bar{x}).$$

Tenendo conto del vincolo aggiunto e della genericità del punto $\bar{x}$:

$$f(\bar{x}) - p(\bar{x}) = f[x_0, x_1, ..., x_m, \bar{x}]\omega_{n+1}(\bar{x}),$$

e quindi abbiamo appena ottenuto la tesi. Cvd.

Corollario

Se la funzione $f(x) \in \mathcal{C}^{(n+1)}$, allora:

$$e(x) = \frac{f^{(n+1)}(\xi_x)}{(n+1)!}\omega_{n+1}(x)$$

con $\xi \in [min\{x_0, x\}, max\{x_n, x\}]$.
Dalla struttura dell'errore di interpolazione si capisce che questo è fondamentalmente guidato da due componenti:

- $f^{(n+1)}(\xi_x)/(n+1)!$ dipende dalle proprietà di regolarità di $f(x)$,
- $\omega_{n+1}(x)$ dipende soltanto dalla scelta delle ascisse di interpolazione.

Si ha pertanto che $\omega_{n+1}(x)$ oscilla quando $x \in [x_0, x_n]$ e si annulla nelle ascisse di interpolazione; ma $|\omega_{n+1}(x)|$ cresce quanto $|x^{n+1}|$ se $x < x_0$ o $x > x_n$. Si può concludere quindi che generalmente $p(x)$ può essere utilizzato in maniera utile per approssimare la funzione di partenza $f(x)$ solo all'interno del range $[x_0, x_n]$.

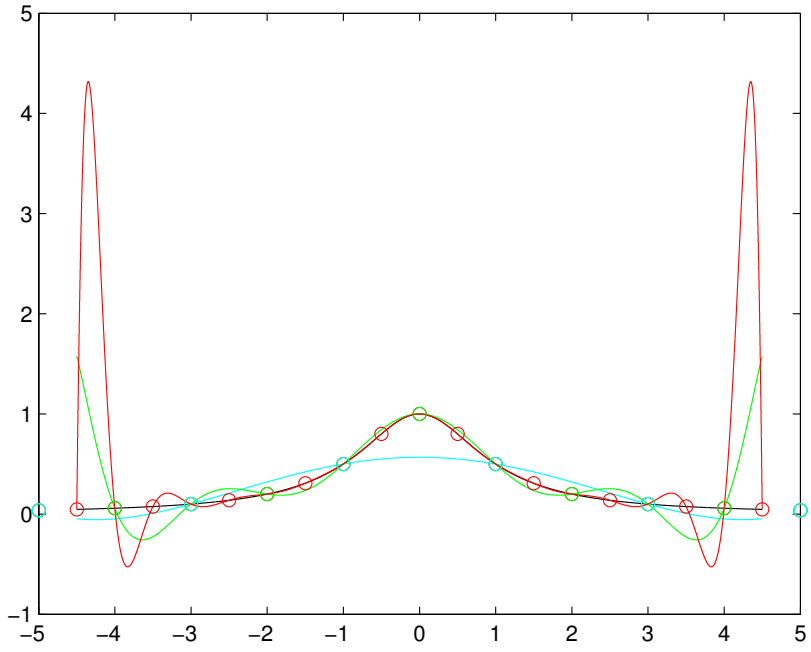

Figura 4.3: Questo grafico mostra l'andamento del polinomio di Newton interpolante la funzione $1/(1+x^2)$ al variare del numero delle ascisse; abbiamo in nero la funzione originaria, in ciano verde e rosso rispettivamente i polinomi interpolanti in: $[-5:2:5]$,$[-5:1:5]$,$[-5:0.5:5]$. Notiamo come l'errore aumenti al crescere del numero dei punti di interpolazione.

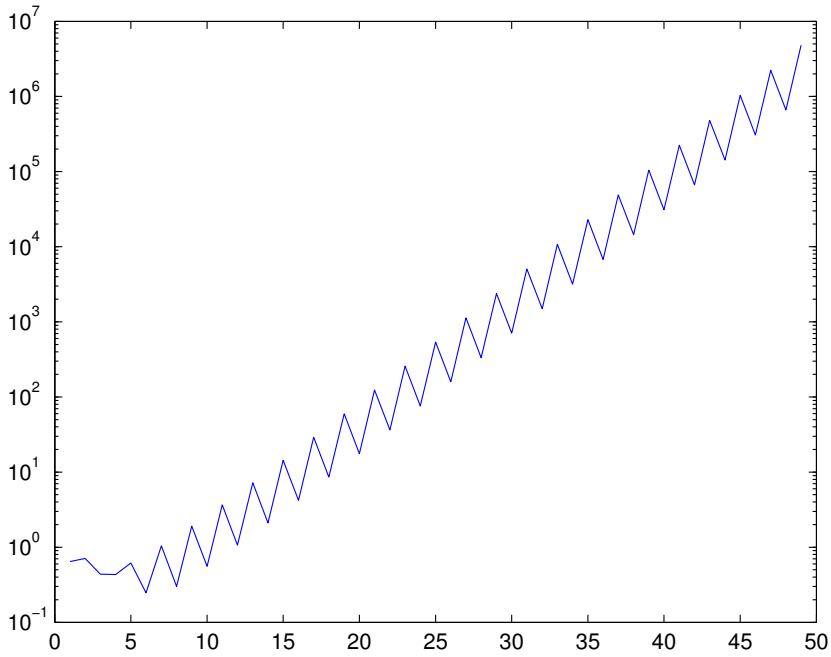

Figura 4.4: Questo grafico mostra l'andamento dell'errore rispetto al numero di punti di interpolazione scelti in modo equidistante nell'intervallo $[-5,5]$ per il polinomio di Newton interpolante la funzione $1/(1+x^2)$. Si nota facilmente che l'errore diverge al crescere del numero dei punti di interpolazione.

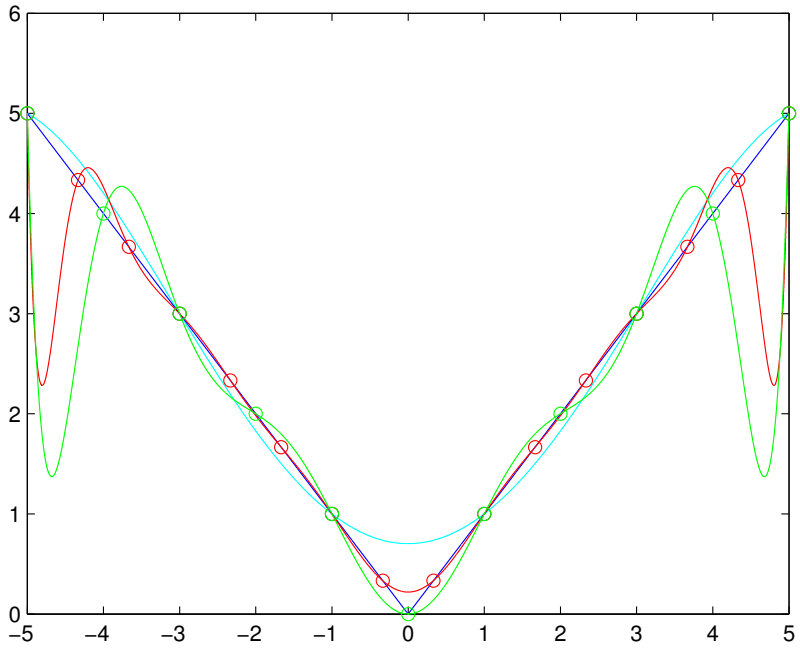

Figura 4.5: Questo grafico mostra l'andamento del polinomio di Newton interpolante la funzione $|x|$ al variare del numero delle ascisse; abbiamo in nero la funzione originaria, in ciano verde e rosso rispettivamente i polinomi interpolanti in 5, 10 e 15 punti equidistanti. Come nel caso della funzione di Runge, l'errore aumenta al crescere del numero dei punti di interpolazione.

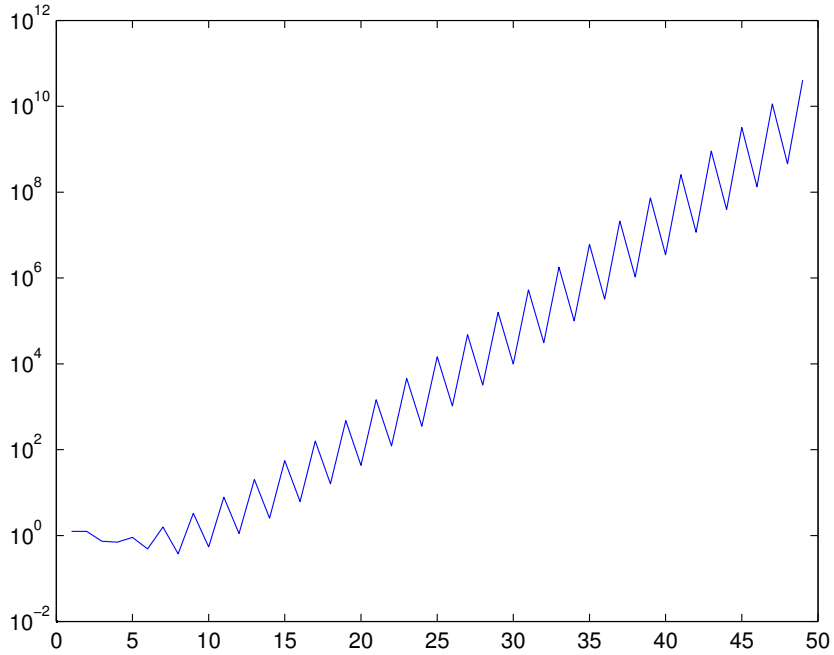

Figura 4.6: Questo grafico mostra l'andamento dell'errore rispetto al numero di punti di interpolazione scelti in modo equidistante nell'intervallo $[-5, 5]$ per il polinomio di Newton interpolante la funzione di Bernstein. Come nel caso della funzione di Runge, l'errore diverge in maniera molto pronunciata al crescere del numero di punti di interpolazione.

4.5 Condizionamento del problema

Per effettuare lo studio del condizionamento del problema occorre per prima cosa identificare quali siano i dati in ingresso per i quali una perturbazione di questi si rifletta sul risultato finale. Per definire il polinomio $p(x)$ ci siamo serviti di un insieme di coppie di numeri (f_i, x_i), delle quali, in generale, possiamo dire che:

- la scelta delle ascisse x_i è arbitraria;
- l'errore commesso sulle ordinate è generalmente più rilevante di quello sulle ascisse (che in genere sono il risultato di misure sperimentali);
- uno stesso insieme di ascisse può essere riutilizzato per definire polinomi che interpolino funzioni differenti.

Possiamo quindi considerare l'insieme dei valori delle ascisse come parametri fissati del problema, e quindi considerare le ordinate gli unici dati in ingresso soggetti ad errore. Consideriamo la funzione $\tilde{f}(x)$ una perturbazione di $f(x)$, e indicheremo con $\tilde{f}_i$ le valutazioni di $\tilde{f}(x_i)$ come già fatto in precedenza con la $f(x)$. Definiamo adesso, espressi nella forma di Lagrange, i due polinomi:

$$p(x) = \sum_{k=0}^{n} f_k L_{kn}(x), \quad \tilde{p}(x) = \sum_{k=0}^{n} \tilde{f}_k L_{kn}(x)$$

che sono rispettivamente il polinomio costruito a partire dai dati esatti e quello invece costruito sui dati perturbati. Confrontandoli si ottiene:

$$|p(x) - \tilde{p}(x)| = \left| \sum_{k=0}^{n} f_k L_{kn}(x) - \sum_{k=0}^{n} \tilde{f}_k L_{kn}(x) \right| = \left| \sum_{k=0}^{n} (f_k - \tilde{f}_k) L_{kn}(x) \right|$$

e per disugualianza triangolare possiamo affermare:

$$\left| \sum_{k=0}^{n} (f_k - \tilde{f}_k) L_{kn}(x) \right| \leq \sum_{k=0}^{n} |f_k - \tilde{f}_k| \cdot |L_{kn}(x)| \leq \left(\sum_{k=0}^{n} |L_{kn}(x)| \right) \max_k |f_k - \tilde{f}_k|$$

$$\equiv \lambda_n(x) \max_k |f_k - \tilde{f}_k|$$

dove $\lambda_n(x)$ è detta funzione di Lebesgue. Per come è stata definita, $\lambda_n(x)$ dipende solo dalla scelta delle ascisse di interpolazione. Definiamo adesso:

$$\|f\| = \max_{a \leq x \leq b} |f(x)|.$$

Abbiamo quindi che:

$$\|p - \tilde{p}\| \leq \|\lambda_n\| \cdot \|f - \tilde{f}\| \equiv \Lambda_n \|f - \tilde{f}\|,$$

dove Λ_n è detta costante di Lebesgue, e dipende soltanto dalla scelta dell'intervallo $[a, b]$ considerato e dalla scelta delle ascisse di interpolazione all'interno di questo.
Abbiamo ottenuto che $\|f - \tilde{f}\|$ è una misura dell'errore sui dati in ingresso, mentre $\|p - \tilde{p}\|$ misura l'errore sul risultato. La costante di Lebesgue appena definita misura pertanto l'amplificazione massima sul risultato dell'errore dei dati in ingresso, e definisce quindi il numero di condizionamento del problema. É utile ricordare che $\Lambda_n \geq O(\log n)$ per $n \to \infty$, ovvero il problema diviene sempre più malcondizionato al crescere di n. Inoltre la scelta di ascisse equidistanti genera una successione $\{\Lambda_n\}$ che diverge in modo esponenziale al crescere di n; ne consegue che questo metodo non è una scelta molto arguta se si vogliono usare valori molto elevati per n.
Vediamo adesso come l'errore di interpolazione ed il condizionamento del problema siano strettamente interconnessi:

Teorema

Sia $f(x)$ una funzione continua in $[a,b]$, allora esiste $p^*(x) \in \Pi_n$ tale che:

$$\|f - p^*\| = \min_{p \in \Pi_n} \|f - p\|.$$

Questo polinomio è detto "polinomio di miglior approssimazione di grado n di $f(x)$ sull'intervallo $[a,b]$".

Teorema

Sia $p^*(x)$ il polinomio di migliore approssimazione di grado n di $f(x)$. Allora si ha che:

$$\|e\| \leq (1 + \Lambda_n)\|f - p^*\|.$$

Dimostrazione:
considerando che $p^*(x) \in \Pi_n$ coincide con il proprio polinomio interpolante sulle ascisse $a \leq x_0 < ... < x_n \leq b$, si ha:

$$\|e\| = \|f - p\| = \|f - p^* + p^* - p\| \leq \|f - p^*\| + \|p^* - p\| \leq (1 + \Lambda_n)\|f - p^*\|$$

come volevasi dimostrare.
Come conseguenza, non è detto che l'errore diminuisca al crescere di n se la costante di Lebesgue cresce esponenzialmente. Per precisare, introduciamo il "modulo di continuità di una funzione" relativo all'intervallo $[a,b]$:

$$\omega(f;h) \equiv sup\{|f(x) - f(y)| : x, y \in [a,b], |x - y| \leq h\}$$

dove $h > 0$ è un parametro assegnato. Possiamo osservare che se $f(x) \in \mathcal{C}^{(1)}$, allora $\omega(f;h) \to 0$ se anche $h \to 0$,e che se $f(x)$ è Lipschitziana con costante L, allora $\omega(f;h) \leq Lh$.

Teorema di Jackson

Il polinomio di approssimazione di una funzione $p(x) = min_{p \in \Pi_n}\|f - p\|$,$f(x) \in \mathcal{C}^{(0)}$, è tale che:

$$\|f - p^*\| \leq \alpha \cdot \omega\left(f; \frac{b-a}{n}\right)$$

in cui la costante α è indipendente da n. Possiamo quindi concluedere che, per una funzione generica vale:

$$\|e\| \leq \alpha(1 + \Lambda_n)\omega\left(f; \frac{b-a}{n}\right).$$

4.6 Ascisse di Chebyshev

Trattiamo adesso la risoluzione del seguente problema:

$$\min_{a \leq x_0 < ... < x_n \leq b} \|\omega_n + 1\| \equiv \min_{a \leq x_0 < ... < x_n \leq b} \max_{a \leq x \leq b} |\omega_{n+1}(x)|$$

che deriva dallo studio del condizionamento del problema, in cui ci si accorge che una scelta oculata delle ascisse di interpolazione deve essere fatta tenedo conto che:

- la costante di Lebesgue Λ_n cresca lentamente rispetto al grado del polinomio interpolante (quanto piu vicino possibile all'andamento logaritmico, che è l'ottimo);

- la quantità $\|\omega_n + 1\|$ sia minimizzata; infatti, supponendo $f(x)$ sufficientemente regolare, è possibile ricavare che:

$$\|e\| \leq \frac{\|f^{(n+1)}\|}{(n+1)!}\|\omega_{n+1}\|$$

dove il fattore $\frac{\|f^{(n+1)}\|}{(n+1)!}$ è indipendente dalla scelta delle ascisse di interpolazione.

Per semplicità, assumiamo che:

$$[a, b] \equiv [-1, 1]$$

In questo modo non si perde in generalità, infatti è sempre possibile costruire una corrispondenza biunivoca tra gli insiemi $Z = \{z : z \in [-1, 1]\}$ e $Y = \{y : y \in [a, b]\}$. Si ha che:

$$y \equiv \left(\frac{a+b}{2} + \frac{b-a}{2}z\right) \in [a, b]$$

infatti se $z = -1, \quad y = a$ e se $z = 1, \quad y = b$; per tutti i valori di z compresi tra -1 ed 1 si ha che alla quantità $\frac{a+b}{2}$ (che indica la metà esatta dell'intervallo $[a, b]$) viene sommata una quantità $-\frac{a+b}{2} \leq \frac{b-a}{2}z \leq \frac{a+b}{2}$ e quindi tutti i valori risultanti devono appartenere all'intervallo $[a, b]$.
Viceversa, si ha che:

$$z \equiv \frac{2y - a - b}{b - a} \in [-1, 1]$$

infatti se $y = a, \quad z = -1$ e se $y = b, \quad z = 1$; per tutti i valori di y compresi tra a e b si ha che $|2y - a - b| \leq |b - a|$ e quindi tutti i valori risultanti appartengono all'intervallo $[-1, 1]$.
Possiamo adesso definirela famiglia di polinomi di Chebyshev di prima specie come:

$$\begin{aligned} &T_0(x) \equiv 1 \\ &T_1(x) = x \\ &T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x), \quad k = 1, 2, ... \end{aligned}$$

Valgono le seguenti proprietà:

1. $T_k(x)$ è un polinomio di grado esatto k
2. Il coefficiente principale di $T_k(x)$ è $2^{k-1}, \quad k = 1, 2, ...$
3. La famiglia di polinomi $\hat{T}_k$,, in cui

$$\hat{T}_0(x) = T_0(x), \quad \hat{T}_k(x) = 2^{1-k}T_k(x), \quad k = 1, 2, ...$$

è una famiglia di polinomi monici di grado k, $k = 1, 2, ...$

4. Se si pone $x = \cos\theta$, con $\theta \in [0, \pi]$, per parametrizzare i punti dell'intervallo [-1,1] rispetto a θ si ottiene:

$$T_k(x) \equiv T_k(\cos\theta) = \cos(k\theta), \quad k = 0, 1, ...$$

Teorema

Gli zeri di $T_k(x)$, tra loro distinti, sono dati da:

$$x_i^{(k)} = \cos\left(\frac{(2i+1)\pi}{2k}\right), \quad i = 0, 1, ..., k-1$$

Per $x \in [-1,1]$, i valori estremi di $T_k(x)$ sono assunti nei punti:

$$\xi_i^{(k)} = \cos\left(\frac{i}{k}\pi\right), \quad i = 0,1,...,k$$

e corrispondono a:

$$T_k(\xi_i^{(k)}) = (-1)^i, \quad i = 0,1,...,k.$$

Si ha quindi che $\|T_k\| = 1$, e $\|\hat{T}_k\| = \|2^{1-k}T_k\| = 1$. Inoltre per $k = 1,2,...$:

$$2^{1-k} = \|\hat{T}_k\| = \min_{\varphi \in \Pi'_k} \|\varphi\|.$$

Segue quindi che, sulle ascisse di interpolazione $a \leq x_0 < ...ax_n \leq b$ per l'intervallo $[-1,1,]$ scelte in questo modo:

$$x_{n-i} = \cos\left(\frac{(2i+1)\pi}{2(n+1)}\right), \quad i = 0,1,...,n$$

si ottiene:

$$\omega_{n+1}(x) = \prod_{i=0}^{n}(x - x_i) \equiv \hat{T}_{n+1}(x)$$

che è la soluzione del problema che ci eravamo posti. Con questa scelta delle ascisse, se la funzione è sufficientemente regolare si ottiene che:

$$\|e\| \leq \frac{\|f^{(n+1)}\|}{(n+1)!2^n}$$

Si ha quindi che la corrispondente costante di Lebesgue vale:

$$\Lambda_n \approx \frac{2}{\pi}\log n$$

che risulta quindi avere una cresctia ottimale anche per un numero di ascisse tendente all'infinito.

4.7 Funzioni *spline*

Abbiamo appena visto come al crescere del grado del polinomio interpolante sia necessario effettuare una scelta delle ascisse di interpolazione che non faccia crescere in maniera importante la costante di Lebesgue; d'altro canto se si mantiene basso il numero n delle ascisse di interpolazione il modulo di continuità di f, $\omega = \left(f; \frac{b-a}{n}\right)$ non può tenedere a zero una volta fissato l'intervallo $[a,b]$. Possiamo però fissare una partizione dell'intervallo originario $\Delta = \{a = x_0 < x_1 < ... < x_n = b\}$, per la quale $h = \max_{i=1,...,n}(x_i - x_{i-1})$ tende a zero se n tende all'infinito, e considerare, per ogni sottointervallo $[x_{i-1}, x_i]$ della partizione, un polinomio di grado m fissato interpolante f negli estremi del sottointervallo. In questo modo si ha che il problema del condizionamento perde importanza in quanto m rimane costante (e quindi anche l'errore), mentre a questo punto vale che $\omega(f;h)$ tende a zero per n che tende all'infinito. Quella che abbiamo appena descritto è una funzione polinomiale a tratti, definiamola più esattamente:

Definizione

$s_m(x)$ è una *spline* di grado m sulla partizione Δ se:

- $s_m(x) \in \mathbb{C}^{(m-1)}$ sull'intervallo $[a,b]$ e

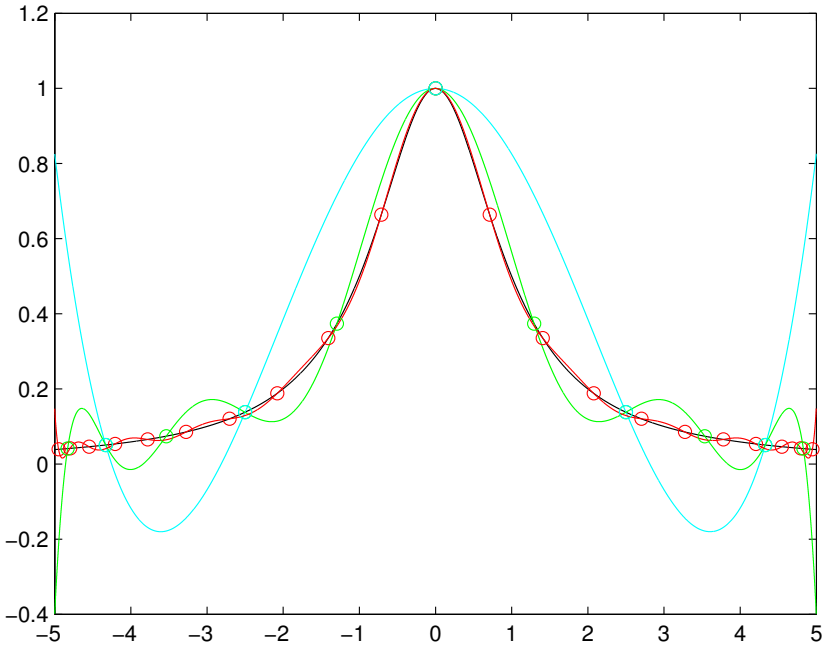

Figura 4.7: Questo grafico mostra l'andamento del polinomio di Newton interpolante la funzione $1/(1+x^2)$ al variare del numero delle ascisse; abbiamo in nero la funzione originaria, in ciano verde e rosso rispettivamente i polinomi interpolanti nelle ascisse di Chebyshev colcolate su 5,11 e 21 punti nell'intervallo $[-5, 5]$. Notiamo come, al contrario di come succedeva con le ascisse equidistanti, l'errore si mantenga molto più basso.

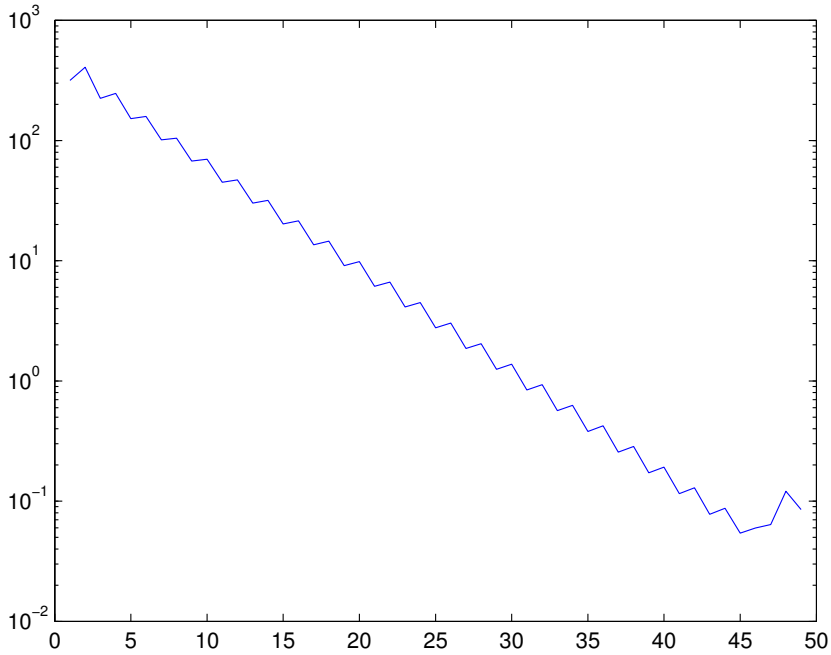

Figura 4.8: Questo grafico mostra l'andamento dell'errore rispetto al numero di punti di interpolazione scelti con il criterio di Chebyshev per il polinomio di Newton interpolante la funzione $1/(1+x^2)$. Si nota facilmente che l'errore diminuisce all'aumentare del numero di punti di interpolazione, contrariamente a quanto accadeva con le ascisse equidistanti.

- $s_m\,\big|_{[x_{i-1},x_i]}(x) \in \Pi_m\ , \quad i = 1, ..., n.$

Se si aggiunge la condizione:

$$s_m(x_i) = f_i, \quad i = 0, 1, ..., n$$

allora la spline è quella interpolante la funzione $f(x)$ nei nodi della partizione Δ.

Teorema

Se $s_m(x)$ è una spline di grado m sulla partizione $\Delta = \{a = x_0 < x_1 < ... < x_n = b\}$, allora $s'_m(x)$ è una spline di grado $m-1$ sulla stessa partizione.
Dimostrazione:
Se $s_m(x) \in \mathbb{C}^{(m-1)}$, allora $s'_m(x) \in \mathbb{C}^{(m-2)}$ e se $s_m|_{[x_{i-1},x_i]}(x) \equiv p_i(x) \in \Pi_m$, allora $p'(x) \in \Pi_{m-1}$.

Teorema

L'insieme delle funzioni spline di grado m definite sulla partizione Δ è uno spazio vettoriale di dimensione $m + n$.

Ne consegue quindi che sono necessarie $m+n$ condizioni indipendenti per individuare univocamente la spline interpolante una funzione su una partizione assegnata ($m+n-1$ se si considera n l'esatto numero di punti di interpolazione); in quanto tipicamente abbiamo in totale $n+1$ condizioni di interpolazione (le coppie (x_i, fi)), queste permettono l'individuazione unica della spline lineare, che coincide con la spezzata congiungente i punti $\{(x_i, f_i)\}_{i=0,...,n}$ in questo modo:

$$s_1|_{[x_{i-1}]}(x) = \frac{(x - x_{i-1})f_i + (x_i - x)f_{i-1}}{x_i - x_{i-1}}, \quad i = 1, ..., n.$$

Per identificare una spline di ordine più elevato è necessaria l'imposizione di oppurtune condizioni aggiuntive.

4.7.1 Spline cubiche

Come detto prima, per ottenere una spline cubica occorre imporre, oltre alle $n + 1$ condizioni di interpolazione, due condizioni aggiuntive, dalla cui scelta si ottengono spline cubiche differenti.

Spline naturale

Questa scelta consiste nell'imporre:

$$s''_3(a) = 0, \quad s''_3(b) = 0$$

Spline completa

In questo caso supponiamo di conoscere i valori di $f'(x)$ negli estremi della partizione, e quindi imponiamo:

$$s'_3(a) = f'(x), \quad s'_3(b) = f'(b)$$

Spline periodica

Se la funzione che stiamo interpolando è periodica e l'intervallo $[a, b]$ contiene un numero finito e intero di periodi, si possono imporre le condizioni:

$$s'_3(a) = s'_3(b), \quad s''_3(a) = s''_3(b).$$

Condizioni not-a-knot

In questo caso imponiamo che uno stesso polinomio in Π_3 costituisca la restrizione della spline sull'intervallo $[x_0, x_1] \cup [x_1, x_2]$, ed un'altro polinomio analogo faccia lo stesso su $[x_{n-1}, x_{n-1}] \cup [x_{n-1}, x_n]$. Le condizioni da imporre saranno quindi:

$$s_3'''|_{[x_0,x_1]}(x_1) = s_3'''|_{[x_1,x_2]}(x_1), \quad s_3'''|_{[x_{n-1},x_{n-1}]}(x_{n-1}) = s_3'''|_{[x_{n-1},x_n]}(x_{n-1})$$

Se si osserva che $s_3'''|_{[x_{i-1},x_i]}(x) \in \Pi_0$, si può esprimere le condizioni suddette come:

$$\frac{s_3''(x_1) - s_3''(x_0)}{x_1 - x_0} = \frac{s_3''(x_2) - s_3''(x_1)}{x_2 - x_1}$$

e

$$\frac{s_3''(x_{n-1}) - s_3''(x_{n-2})}{x_{n-1} - x_{n-2}} = \frac{s_3''(x_n) - s_3''(x_{n-1})}{x_n - x_{n-1}}.$$

4.7.2 Calcolo di una spline cubica

Andiamo adesso ad approssimare una funzione $f(x)$ definita su un'intervallo $[a, b]$ con una spline cubica, prendendo in esame una partizione dell'intervallo $\Delta = \{a = x_0 < ... < x_n = b\}$. Osserviamo come se $s_3(x)$ è una spline cubica, allora $s_3'(x)$ è una spline quadratica e $s_3''(x)$ una lineare. Vale quindi che:

$$s_3''|_{[x_{i-1},x_i]}(x) = \frac{(x - x_{i-1})m_i + (x_i - x)m_{i-1}}{h_i}, \quad x \in [x_{i-1}, x_i]$$

dove:

$$m_i \equiv s_3''(x_i), \quad i = 0, 1, ..., n,$$
$$h_i = x_i - x_{i-1}, \quad i = 1, 2, ..., n.$$

Integrando questa equazione si ottiene :

$$s_3'|_{[x_{i-1},x_i]}(x) = \frac{(x - x_{i-1})^2 m_i + (x_i - x)^2 m_{i-1}}{2h_i} + q_i, \quad x \in [x_{i-1}, x_i]$$

dove q_i è una costante di integrazione, e integrando ancora una volta si ottiene:

$$s_3|_{[x_{i-1},x_i]}(x) = \frac{(x - x_{i-1})^3 m_i + (x_i - x)^3 m_{i-1}}{6h_i} + q_i(x - x_{i-1}) + r_i, \quad x \in [x_{i-1}, x_i]$$

dove r_i è una costante di integrazione. Imponiamo adesso i vincoli di interpolazione, ovvero che $s_3(x_i) = f_i, \quad i = 0, 1, ..., n$, per ricavare r_i e q_i:

$$s_3(x_i) = \frac{h_i^2}{6} m_i + q_i h_i + r_i = f_i$$
$$s_3(x_{i-1}) = \frac{h_i^2}{6} m_{i-1} + r_i = f_{i-1}$$

da cui:

$$r_i = f_{i-1} - \frac{h_i^2}{6} m_{i-1}$$
$$q_i = \frac{f_i - f_{i-1}}{h_i} - \frac{h_i}{6}(m_i - m_{i-1})$$

Abbiamo quindi che :

$$s_3'|_{[x_{i-1},x_i]}(x) = \frac{(x - x_{i-1})^2 m_i + (x_i - x)^2 m_{i-1}}{2h_i} + \frac{f_i - f_{i-1}}{h_i} - \frac{h_i}{6}(m_i - m_{i-1}),$$

sempre con $x \in [x_{i-1}, x_i]$. Imponiamo adesso la continuità di $s_3'(x)$, tenendo conto che:

- bisogna imporre la continuità nei punti di intersezione delle singole parti della partizione Δ: $x_i = [x_{i-1}, x_i] \cap [x_i, x_{i+1}]$;
- $\frac{f_i - f_{i-1}}{h_i} = f[x_{i-1}, x_i]$

quindi:

$$\begin{array}{rcc} s_3'|_{[x_{i-1},x_i]}(x_i) & = & s_3'|_{[x_i,x_{i+1}]}(x_i) \\ \frac{h_i}{2}m_i - \frac{h_i}{6}(m_i - m_{i-1}) + f[x_{i-1}, x_i] & = & -\frac{h_{i+1}}{2}m_i + f[x_i, x_{i+1}] - \frac{h_{i+1}}{6}(m_{i+1} - m_i) \\ h_i m_{i-1} + m_i(2h_i + 2h_{i+1}) + h_{i+1}m_{i+1} & = & 6(f[x_i, x_{i+1}] - f[x_{i-1}, x_i]) \end{array}$$

e dividendo membro a membro per $h_i + h_{i+1} (= x_{i+1} - x_{i-1})$:

$$\varphi_i = \frac{h_i}{h_i + h_{i+1}}, \quad \xi_i = \frac{h_{i+1}}{h_i + h_{i+1}}, \quad i = 1, ..., n-1.$$

e quindi:

$$\varphi_i m_{i-1} + 2m_i + \xi_i mi + 1 = 6f[x_{i-1}, x_i, x_{i+1}], \quad i = 1, ..., n-1.$$

Si osserva che $\varphi_i, \xi_i > 0$ e $\varphi_i + \xi_i = 1$ per $i = 1, ..., n-1$.
Poniamoci adesso nel caso delle spline naturali, e imponiamo quidni la condizione $m_0 = m_n = 0$. Si ottiene immediatamente il sistema:

$$\begin{pmatrix} 2 & \xi_1 & & & \\ \varphi_2 & 2 & \xi_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \xi_{n-2} \\ & & & \varphi_{n-1} & 2 \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ \vdots \\ \vdots \\ m_{n-1} \end{pmatrix} = 6 \begin{pmatrix} f[x_0, x_1, x_2] \\ f[x_1, x_2, x_3] \\ \vdots \\ \vdots \\ f[x_{n-2}, x_{n-1}, x_n] \end{pmatrix}$$

Notiamo come questa matrice goda di diverse proprietà:

- i coefficienti sono tutti positivi;
- è diagonale dominante per righe: questo significa che esiste la fattorizzazione LU senza pivoting ed è sempre ben condizionata;
- la partizione uniforme è una buona scelta per la risoluzione di questo tipo di problema
- il costo della fattorizzazione LU è lineare con la dimensione della matrice.

Condizioni not-a-knot

Andiamo adesso ad imporre le condizioni not-a-knot:

$$\frac{s_3''(x_1) - s_3''(x_0)}{x_1 - x_0} = \frac{s_3''(x_2) - s_3''(x_1)}{x_2 - x_1}; \quad \frac{s_3''(x_n) - s_3''(x_{n-1})}{x_n - x_{n-1}} = \frac{s_3''(x_{n-1}) - s_3''(x_{n-2})}{x_{n-1} - x_{n-2}}$$

che significa porre:

$$\frac{m_2 - m_1}{h_2} = \frac{m_1 - m_0}{h_1}, \quad \frac{m_n - m_{n-1}}{h_n} = \frac{n_{n-1} - n_{n-2}}{h_{n-1}}$$

da cui si ricava:

$$\begin{array}{rcl} m_2 h_1 - m_1 h_2 & = & h_2 m_1 - h_2 m_0 \\ \frac{m_0 h_2}{h_1 + h_2} - \frac{m_1(h_1 + h_2)}{h_1 + h_2} + \frac{h_1 m_2}{h_1 + h_2} & = & 0 \\ m_0 \underbrace{\frac{h_2}{h_1 + h_2}}_{\xi_1} - m_1 + \underbrace{\frac{h_1}{h_1 + h_2}}_{\varphi_1} m_2 & = & 0 \end{array}$$

in modo analogo si ricava che:

$$\underbrace{\frac{h_n}{h_n+h_{n-1}}}_{\xi_{n-1}} m_{n-2} - m_{n-1} + \underbrace{\frac{h_{n-1}}{h_n+h_{n-1}}}_{\varphi_{n-1}} = 0$$

che da origine alla matrice:

$$\begin{pmatrix} \xi_1 & -1 & \varphi_1 & & \\ \varphi_1 & 2 & \xi_1 & & \\ & \ddots & \ddots & \ddots & \\ & & \varphi_{n-1} & 2 & \xi_{n-1} \\ & & \xi_{n-1} & -1 & \varphi_{n-1} \end{pmatrix} \begin{pmatrix} m_0 \\ m_1 \\ \vdots \\ m_{n-1} \\ m_n \end{pmatrix} = 6 \begin{pmatrix} 0 \\ f[x_0,x_1,x_2] \\ f[x_1,x_2,x_3] \\ \vdots \\ f[x_{n-2},x_{n-1},x_n] \\ 0 \end{pmatrix}$$

Sommando la seconda riga alla prima e la penultima all'ultima si ottiene:

$$\begin{pmatrix} 1 & 1 & \underline{1} & & \\ \varphi_1 & 2 & \xi_1 & & \\ & \ddots & \ddots & \ddots & \\ & & \varphi_{n-1} & 2 & \xi_{n-1} \\ & & \underline{1} & 1 & 1 \end{pmatrix} \begin{pmatrix} m_0+m_1 \\ m_1 \\ \vdots \\ m_{n-1} \\ m_n+m_{n-1} \end{pmatrix} = 6 \begin{pmatrix} f[x_0,x_1,x_2] \\ f[x_0,x_1,x_2] \\ f[x_1,x_2,x_3] \\ \vdots \\ f[x_{n-2},x_{n-1},x_n] \\ f[x_{n-2},x_{n-1},x_n] \end{pmatrix}$$

che ancora non è una matrice tridiagonale a causa degli elementi sottolineati. Possiamo però sottrarre la prima colonna dalla seconda e dalla terza e l'ultima dalla penultima e terzultima per ottenere finalmente una matrice tridiagonale. Per fare questo ci serviremo di una matrice siffatta:

$$T = \begin{pmatrix} 1 & -1 & -1 & & & & \\ 0 & 1 & 0 & & & & \\ 0 & 0 & 1 & & & & \\ & & & \ddots & & & \\ & & & & 1 & 0 & 0 \\ & & & & 0 & 1 & 0 \\ & & & & -1 & -1 & 1 \end{pmatrix}; \quad T^{-1} = \begin{pmatrix} 1 & 1 & 1 & & & & \\ 0 & 1 & 0 & & & & \\ 0 & 0 & 1 & & & & \\ & & & \ddots & & & \\ & & & & 1 & 0 & 0 \\ & & & & 0 & 1 & 0 \\ & & & & 1 & 1 & 1 \end{pmatrix};$$

Quello che andremo a svolgere invece di $A\mathbf{m} = \mathbf{b}$ diviene $ATT^{-1}\mathbf{m} = \mathbf{b}$, e la matrice AT è quindi:

$$\begin{pmatrix} 1 & 0 & & & & & \\ \varphi_1 & 2-\varphi_1 & \xi_1-\varphi 1 & & & & \\ & \varphi_2 & 2 & \xi_2 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \varphi_{n-2} & 2 & \xi_{n-2} & \\ & & & & \varphi_{n-1}-\xi_{n-1} & 2-\xi_{n-1} & \xi_{n-1} \\ & & & & & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} m_0+m_1+m_2 \\ m_1 \\ \vdots \\ m_{n-1} \\ m_n+m_{n-1}+m_{n-2} \end{pmatrix} = 6 \begin{pmatrix} f[x_0,x_1,x_2] \\ f[x_0,x_1,x_2] \\ \vdots \\ f[x_{n-2},x_{n-1},x_n] \\ f[x_{n-2},x_{n-1},x_n] \end{pmatrix}$$

che è tridiagonale e fattorizzabile LU in quanto ha tutti i minori principali non nulli.

Spline completa

Per avere una spline completa invece dobbiamo imporre:

$$s_3'(a) = f'(a), \quad s_3'(b) = f'(b)$$

ovvero (considerando per comodità gli indici che partono da 1 anziché da 0 ed arrivano ad n, in accordo con quanto utilizzato nelle implementazioni):

$$s_3'(a) \equiv s_3'(x_1) = \frac{(x_1 - x_1)^2 m_2 - (x_2 - x_1)^2 m_1}{2(x_2 - x_1)} + \frac{f_2 - f_1}{x_2 - x_1} + \frac{(x_2 - x_1)}{6}(m_2 - m_1) = f'(a);$$

$$s_3'(b) \equiv s_3'(x_n) = \frac{(x_n - x_{n-1})^2 m_n - (x_n - x_n)^2 m_{n-1}}{2(x_n - x_{n-1})} + \frac{f_n - f_{n-1}}{x_n - x_{n-1}} + \frac{(x_n - x_{n-1})}{6}(m_n - m_{n-1}) = f'(b);$$

da cui si ottiene:

$$-3h_1 m_1 + h_1(m_2 - m_1) = 6f'(a) - 6f[x_1, x_2]; \quad 3h_{n-1} m_n + h_{n-1}(m_n - m_{n-1}) = 6f'(b) - 6f[x_n, x_{n-1}];$$

ovvero:

$$-4h_1 m_1 + h_1 m_2 = 6(f'(a) - f[x_1, x_2]); \quad 4h_{n-1} m_n - h_{n-1} m_{n-1} = 6(f'(b) - f[x_{n-1}, x_n]).$$

Si ricava quindi la matrice:

$$\begin{pmatrix} -4h_1 & h_1 & & & & \\ \varphi_1 & 2 & \xi_1 & & & \\ & \varphi_2 & 2 & \xi_2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \varphi_{n-1} & 2 & \xi_{n-1} \\ & & & & -h_{n-1} & 4h_{n-1} \end{pmatrix} \begin{pmatrix} m_1 \\ \vdots \\ \vdots \\ \vdots \\ m_n \end{pmatrix} = 6 \begin{pmatrix} f'(a) - f[x_1, x_2] \\ f[x_1, x_2] \\ \vdots \\ \vdots \\ f[x_1, x_2] \\ f'(b) - f[x_{n-1}, x_n] \end{pmatrix}$$

che ancora una volta è tridiagonale con tutti i minori principali non nulli, e pertanto fattorizzabile LU.

Spline periodica

Per calcolare la spline cubica periodica invece dobbiamo imporre:

$$s_3'(a) = s_3'(b); \quad s_3''(a) = s_3''(b).$$

Sostituendo nelle formule (come nel caso precedente per semplicità sono stati usati gli indici Matlab) andiamo a ricavare:

Periodicità della derivata prima ($s_3'(a) = s_3'(b)$):

$$-\frac{(x_2 - x_1)m_1}{2} + f[x_1, x_2] - \frac{h_1}{6}(m_2 - m_1) = \frac{(x_n - x_{n-1})m_n}{2} + f[x_n, x_{n-1}] - \frac{h_{n-1}}{6}(m_n - m_{n-1})$$

$$-3h_1 m_1 - 3h_{n-1} m_n - h_1(m_2 - m_1) + h_{n-1}(m_n - m_{n-1}) = 6(f[x_n, x_{n-1}] - f[x_1, x_2])$$

da cui otteniamo quindi la prima condizione:

$$-2h_1 m_1 - 2h_{n-1} m_n - h_1 m_2 - h_{n-1} m_{n-1} = 6(f[x_n, x_{n-1}] - f[x_1, x_2]).$$

Periodicità della derivata seconda ($s_3''(a) = s_3''(b)$):

$$\frac{(x_1 - x_1)m_2 + (x_2 - x_1)m_1}{h_1} = \frac{(x_n - x_{n-1})m_n + (x_n - x_n)m_{n-1}}{h_{n-1}}$$

$$\frac{h_1 m_1}{h_1} = \frac{h_{n-1}m_n}{h_{n-1}} \quad \text{da cui la condizione} \quad m_1 = m_n$$

Andiamo quindi a costruire la matrice relativa al sistema lineare risultante:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \dots & 0 & -1 \\ \varphi_1 & 2 & \xi_1 & 0 & 0 & \dots & \dots & 0 \\ 0 & \varphi_2 & 2 & \xi_2 & 0 & \dots & \dots & 0 \\ \vdots & & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & & \varphi_{n-2} & 2 & \xi_{n-2} & 0 \\ 0 & & & & & \varphi_{n-1} & 2 & \xi_{n-1} \\ -2h_1 & -h_1 & 0 & \dots & \dots & 0 & -h_{n-1} & -2h_{n-1} \end{pmatrix} \begin{pmatrix} m_1 \\ \vdots \\ \vdots \\ \vdots \\ m_n \end{pmatrix}$$

$$= 6 \begin{pmatrix} 0 \\ f[x_1, x_2] \\ \vdots \\ \vdots \\ \vdots \\ f[x_n, x_{n-1}] \\ f[x_n, x_{n-1}] - f[x_1 - x_2] \end{pmatrix}$$

4.8 Approssimazione polinomiale ai minimi quadrati

Il problema che ci poniamo in questa sezione è quello di determinare il polinomio che meglio approssima una serie di dati sperimentali, i quali tipicamente sono in sovrabbondanza ma soggetti ad errori. Vogliamo quindi determinare un polinomio di grado m:

$$y = \sum_{k=0}^{m} a_k x^k$$

che meglio approssimi i dati:

$$(x_i, y_i), \quad i = 0, 1, ..., n, \quad n \geq m$$

dove le ascisse non sono necessariamente tutte distinte, anche se dobbiamo imporre che nella determinazione di un polnomio in Π_m, almeno $m + 1$ lo siano. Definiamo quindi i vettori **f** e **y** rispettivamente dei valori osservati e previsti:

$$\mathbf{f} = \begin{pmatrix} f_0 \\ \vdots \\ f_n \end{pmatrix} \in \mathbb{R}^{n+1}; \quad \mathbf{y} = \begin{pmatrix} y_0 \\ \vdots \\ y_n \end{pmatrix} \in \mathbb{R}^{n+1}, \quad y_i = p_m(x_i) = \sum_{k=0}^{m} a_k x_i^k;$$

rimane da determinare il vettore $\mathbf{a} = (a_0, a_1, ..., a_m)^T$ che minimizza la quantità:

$$\|f - y\|_2^2 = \sum_{i=0}^{n} |f_i - y_i|^2$$

ovvero:

$$p_m(x_i) = (x_i^0 x_i^1 ... x_i^m) \begin{pmatrix} a_0 \\ \vdots \\ a_m \end{pmatrix}$$

Possiamo esprimere quindi il problema in questa forma:

$$\mathbf{y} = V\mathbf{a}; \quad V = \begin{pmatrix} x_0^0 & x_0^1 & \cdots & x_0^m \\ \vdots & & & \vdots \\ x_n^0 & x_n^1 & \cdots & x_n^m \end{pmatrix} \in \mathbb{R}^{n+1 \times m+1}$$

Si ha che:

$$\min_{p_m \in \Pi_m} \|\mathbf{f} - \mathbf{y}\|_2^2 = \min_{\mathbf{a} \in \mathbb{R}^{m+1}} \|V\mathbf{a} - \mathbf{f}\|_2^2$$

Teorema

La matrice V ha rango $m+1$.
Dimostrazione:
Se ci si limita a considerare $m+1$ righe corrispondenti ad ascisse distinte, esse formano una matrice di Vandermonde nonsingolare, e quindi il rango è appunto $m+1$.
Il problema diventa quindi quello di risolvere, nel senso dei minimi quadrati, il sistema sovradeterminato:

$$V\mathbf{a} = \mathbf{y}$$

Possiamo quindi considerare:

$$V = QR = Q \begin{pmatrix} \hat{R} \\ 0 \end{pmatrix}, \quad \hat{R} \in \mathbb{R}^{m+1 \times m+1}, \quad Q^T Q = I_{r+1}$$

e quindi:

$$\min_{p \in \Pi_m} \|\mathbf{f} - \mathbf{y}\|_2^2 = \min_{\mathbf{a} \in \mathbb{R}^{m+1}} \|V\mathbf{a} - \mathbf{f}\|_2^2 = \min_{\mathbf{a} \in \mathbb{R}^{m+1}} \|QR\mathbf{a} - \mathbf{f}\|_2^2 = \|Q(R\mathbf{a} - \underbrace{Q^T\mathbf{f}}_{g})\|_2^2$$

$$= \|R\mathbf{a} - \mathbf{g}\| = \left\| \begin{pmatrix} \hat{R} \\ 0 \end{pmatrix} \mathbf{a} - \begin{pmatrix} \mathbf{g}_1 \\ \mathbf{g}_2 \end{pmatrix} \right\|_2^2 = \left\| \begin{pmatrix} \hat{R}\mathbf{a} - \mathbf{g}_1 \\ -\mathbf{g}_2 \end{pmatrix} \right\| = \|\hat{R}\mathbf{a} - \mathbf{g}_1\|_2^2 + \|\mathbf{g}_2\|_2^2 \Rightarrow \ \hat{R}\mathbf{a} = \mathbf{g}_1$$

Ne deriva quindi che la soluzione di questa equazione, $\mathbf{a} = \hat{R}^{-1}\mathbf{g}_1$ esiste ed è unica, ed è quindi tale anche il polinomio di approssimazione ai mini quadrati. Notiamo che nel caso particolare in cui $m = n$, il polinomio di approssimazione ai minini quadrati coincide esattamente con quello interpolante sulle stesse ascisse; in tal caso infatti il vettore residuo $\mathbf{g}_2$ è il vettore vuoto e il polinomio quindi interpola tutti i valori assegnati.

Osservazione

Può succedere di aver bisogno di minimizzare, invece della norma del vettore dei residui, un vettore pesato $\|D(V\mathbf{a} - \mathbf{f}\|_2^2$ che permetta appunto di dare un'importanza maggiore o minore a certe misurazione rispetto alle altre. Definiamo quindi:

$$D = \begin{pmatrix} w_0 & & \\ & \ddots & \\ & & w_n \end{pmatrix}, \quad w_i > 0, i = 0, ..., m.$$

abbiamo quindi che:

$$\mathbf{r} = \mathbf{y} - \mathbf{f} = \begin{pmatrix} p_0(x_0) - f_0 \\ \vdots \\ p_m(x_n) - f_n \end{pmatrix}.$$

Quindi il problema:

$$\|\mathbf{r}\|_2^2 = \sum_{i=0}^{n} |r_i|^2$$

diventa:

$$\|D\mathbf{r}\|_2^2 = \sum_{i=0}^{n} |r_i| w_i^2 = \sum_{i=0}^{n} w_i^2 (p_m(x_i) - f_i)^2.$$

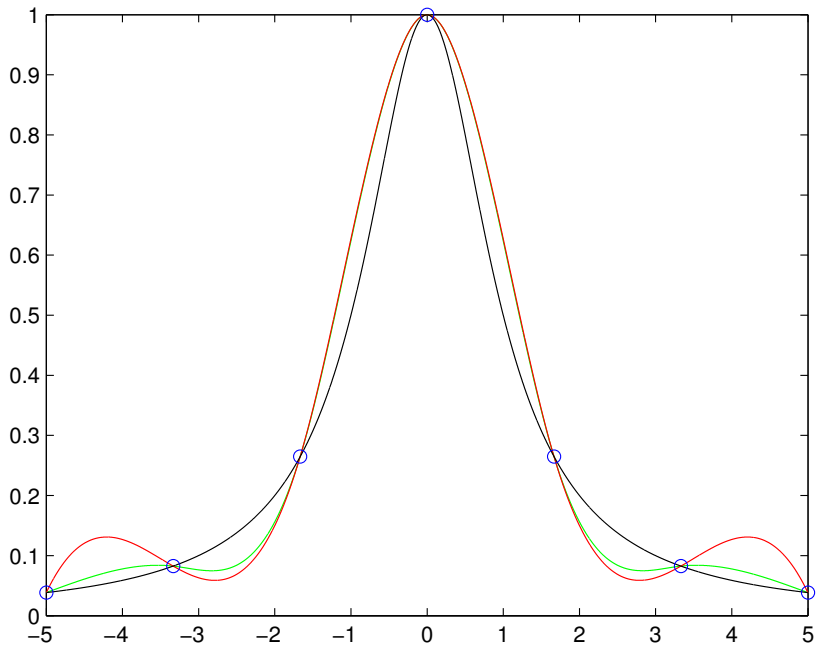

Figura 4.9: Questo grafico mostra l'andamento della spline cubica naturale (verde) e di quella not-a-knot (rossa) interpolanti la funzione $1/(1+x^2)$ in sette punti equidistanti nell'intervallo $[-5, 5]$

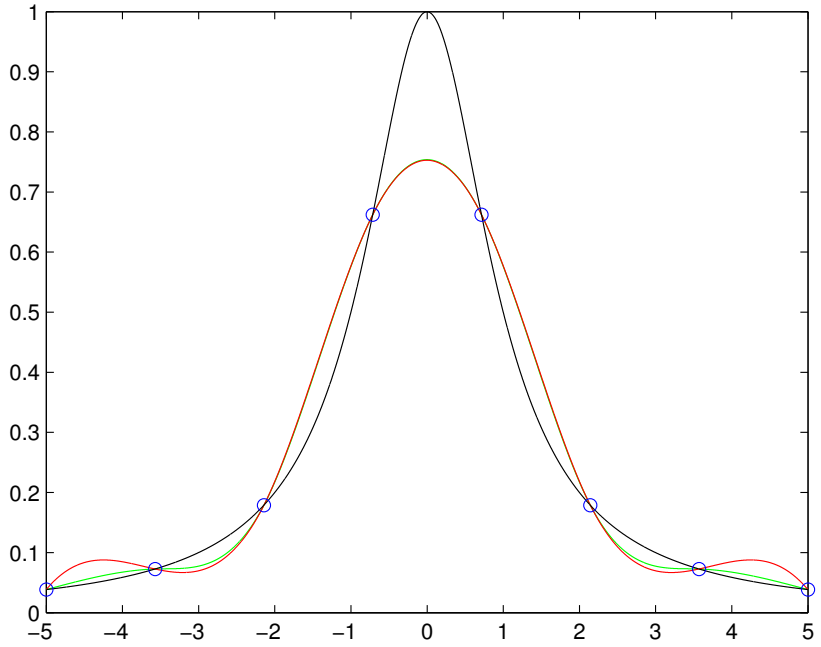

Figura 4.10: Questo grafico mostra invece l'andamento delle stesse spline di cui sopra ma interpolanti la funzione in otto punti equidistanti nello stesso intervallo.

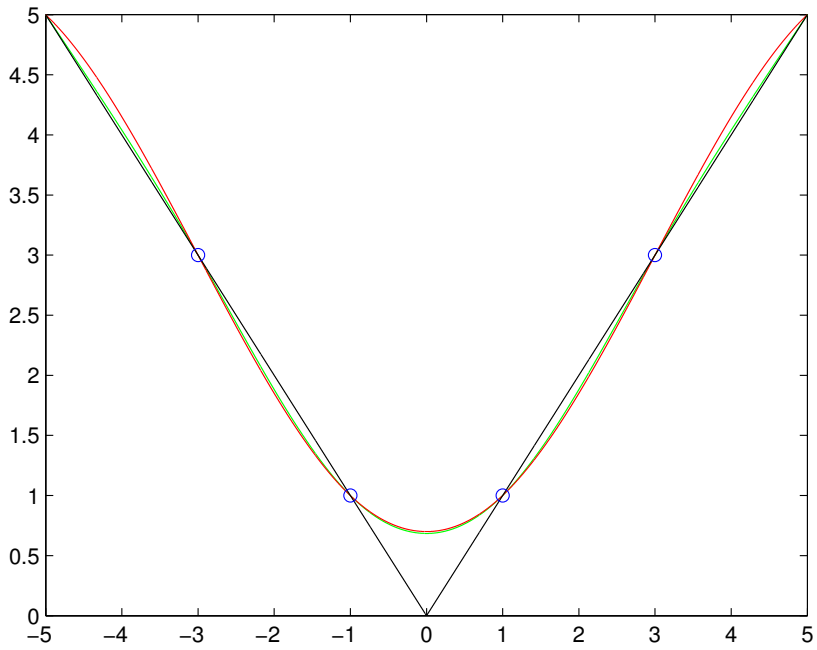

Figura 4.11: Questo grafico mostra l'andamento della spline cubica naturale (verde) e di quella not-a-knot (rossa) interpolanti la funzione $|x|$) in sette punti equidistanti nell'intervallo $[-5, 5]$

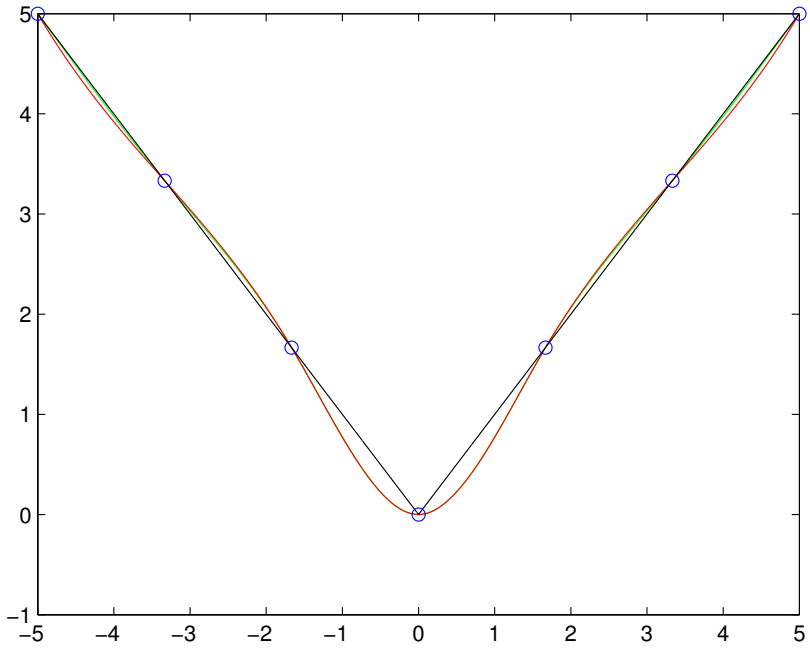

Figura 4.12: Questo grafico mostra invece l'andamento delle stesse spline di cui sopra ma interpolanti la funzione in otto punti equidistanti nello stesso intervallo.

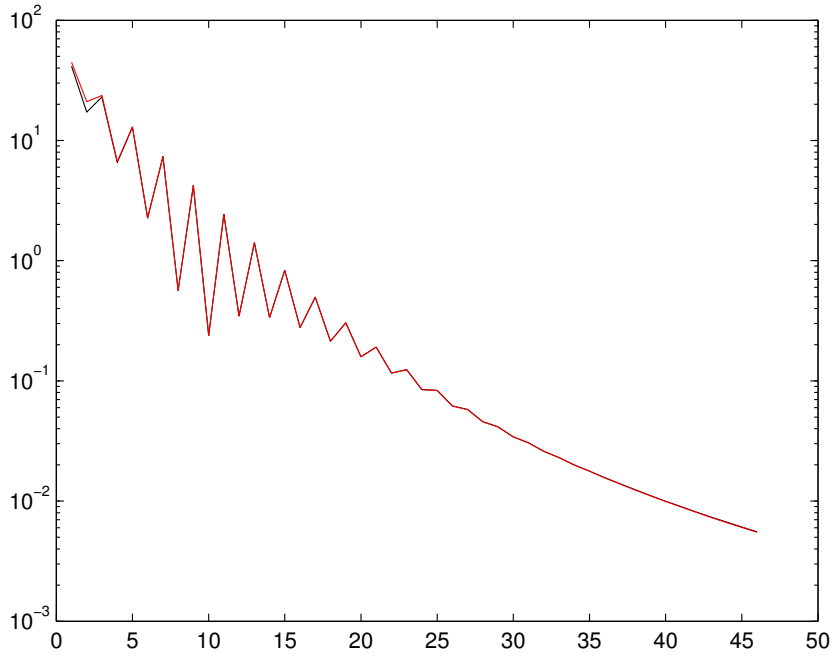

Figura 4.13: Questo grafico mostra l'andamento dell'errore di interpolazione per le funzioni spline interpolanti la funzione $1/(1+x^2)$ all'aumentare del numero di punti di interpolazione. Notiamo come l'errore vari molto quando il numero di punti di interpolazione passa da dispari a pari (per valori piccoli): questo è causato dal fatto che nel caso di numeri pari viene saltato il punto centrale dell'intervallo in cui la funzione assume valore massmo.

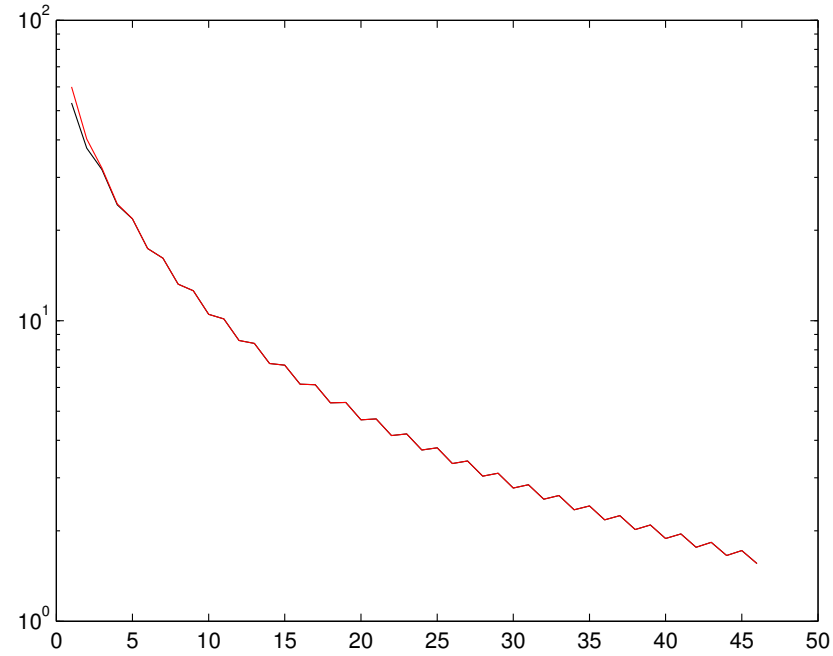

Figura 4.14: Questo grafico mostra l'andamento dell'errore di interpolazione per le funzioni spline interpolanti la funzione $|x|$ all'aumentare del numero di punti di interpolazione.

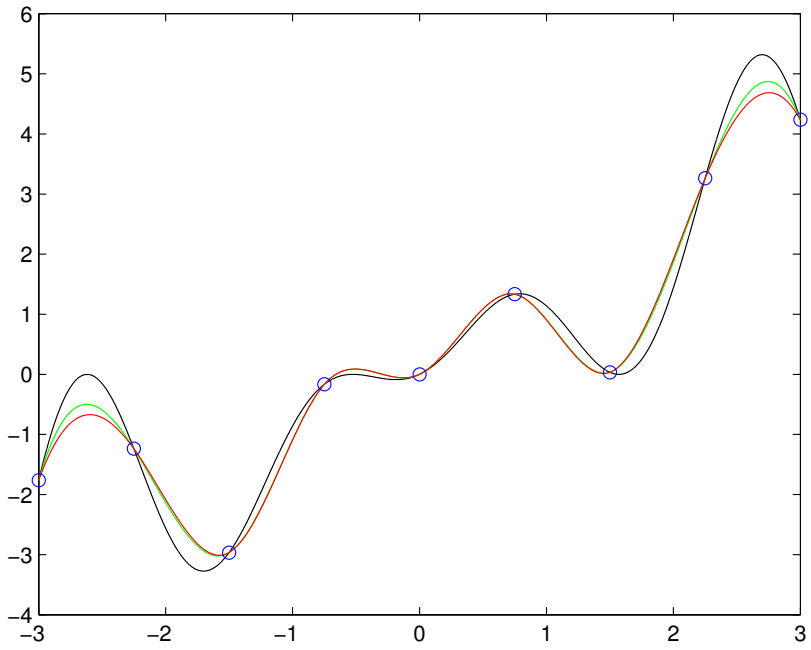

Figura 4.15: Questo grafico mostra l'andamento di una spline completa (rosso) confrontata con quello di una spline not-a-knot (verde) nell'interpolare la funzione $xsin(3x) + x$.

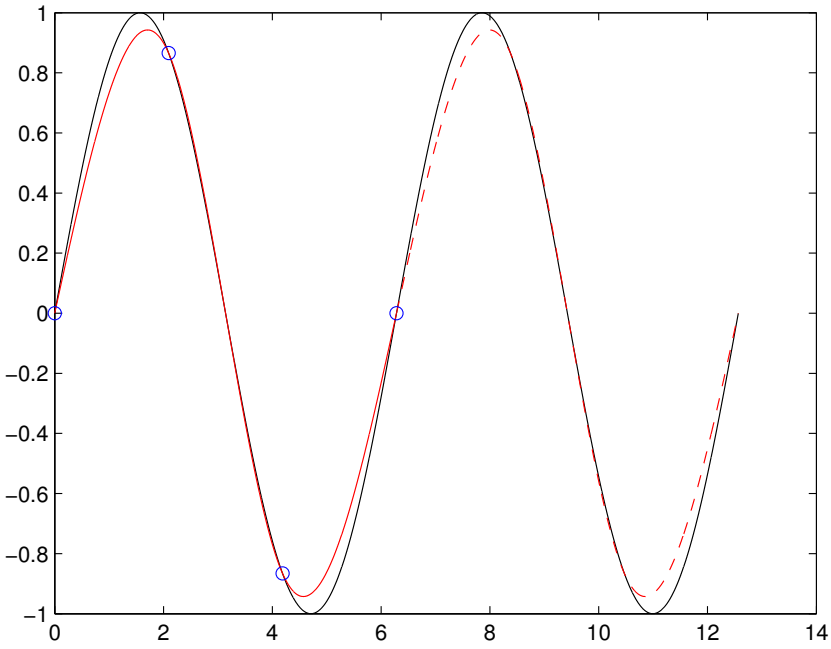

Figura 4.16: Questo grafico mostra l'andamento di una spline periodica interpolante la funzione $sin(x)$ in 4 punti equidistanti.

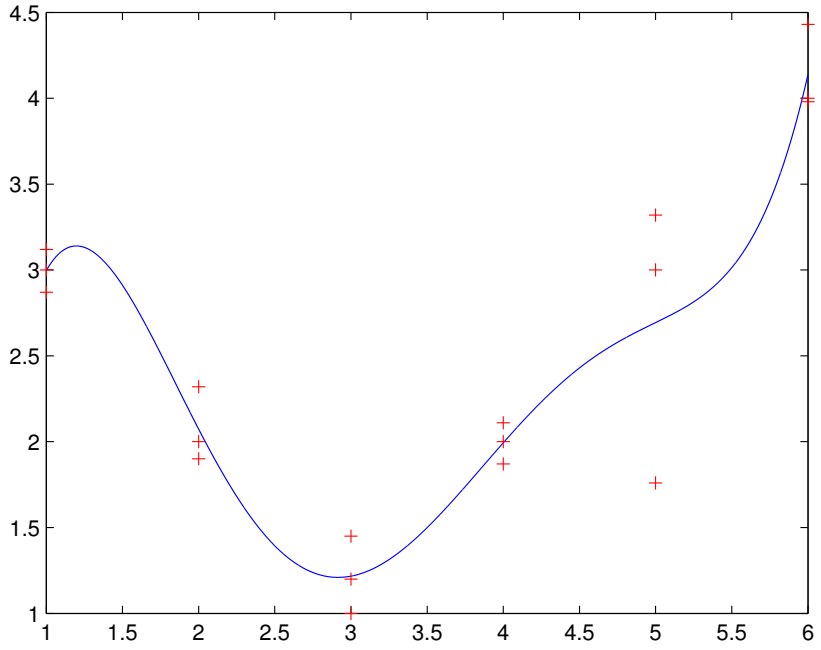

Figura 4.17: Esempio di approssimazione ai minimi quadrati di una serie di dati.

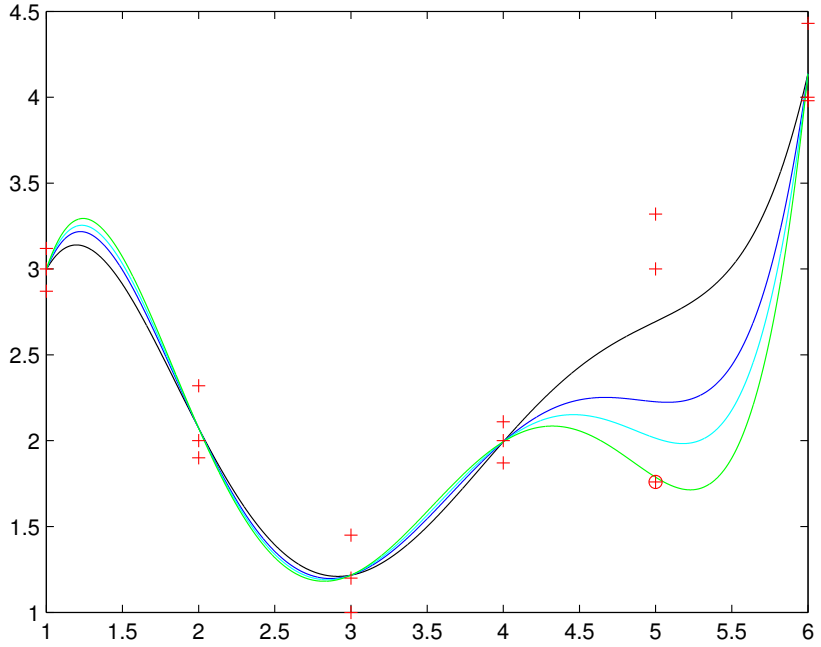

Figura 4.18: Esempio di come cambia l'approssimazione all'aumentare progressivo del perso del punto evidenziato. In ordine crescente di peso le funzioni risultanti sono di colore nero (1), blu(2), ciano(3), verde(10).

Capitolo 5

Formule di quadratura

Consideriamo adesso il problema di voler calcolare, in modo approssimato, il valore di un integrale definito:

$$I(f) = \int_a^b f(x)dx, \quad a < b$$

con $[a, b]$ intervallo limitato, ed $f(x)$ continua su tale intervallo. Per fare ciò andremo a calcolare l'integrale di una funzione polinomiale (o polinomiale a tratti) che approssimi $fx)$, cosa che siamo in grado di fare in modo semplice e senza approssimazioni. Quello che adremo a fare sarà quindi calcolare:

$$I(f) = \int_a^b f(x)dx \approx \int_a^b \varphi(x)dx, \quad \varphi(x) \approx f(x)$$

Studiamo adesso il condizionamento del problema:

$$|I(f) - I(\varphi)| = \left| \int_a^b (f(x) - \varphi(x))dx \right|$$

questa quantità può essere maggiorata con:

$$|I(f) - I(\varphi)| \leq \int_a^b |f(x) - \varphi(x)|dx \leq \max_{x\in[a,b]} |f(x) - \varphi(x)| \int_a^b dx$$

$$\leq \|f - \varphi\| \int_a^b 1dx = (b - a)\|f - \varphi\|.$$

Il fattore $(b-a)$ rappresenta quindi il fattore di amplificazione dell'errore sul risultato, e quindi:

$$k = b - a$$

definisce il numero di condizinamento del problema.

5.1 Formule di Newton-Cotes

Consideriamo l'approssimazione di $f(x)$ fornita dal polinomio interpolante su $n + 1$ ascisse equidistanti:

$$\begin{aligned} p(x_i) &= f(x_i) \equiv f_i, \quad i = 0, 1, ..., n; \\ x_i &= a + ih, \qquad h = \frac{b-a}{n} \end{aligned}$$

Se si considera la forma di Lagrange del polinomio interpolante si ottiene:

$$I(f) \approx \int_a^b \sum_{k=0}^{n} f_k L_{kn}(x)dx = \sum_{k=0}^{n} f_k \int_a^b L_{kn}(x)dx = \sum_{k=0}^{n} f_k \int_a^b \prod_{j=0, j\neq k}^{n} \frac{x - x_j}{x_i - x_j}dx$$

ponendo $x = a + th, \quad t = 0, ..., n$, si ottiene:

$$= \frac{b-a}{n} \sum_{k=0}^{n} \left(f_k \int_0^n \prod_{j=0, j\neq k}^{n} \frac{a + th - (a + jh)}{a + kh - (a + jh)}dt \right) = h \sum_{k=0}^{n} \left(f_k \int_0^n \prod_{j=0, j\neq k}^{n} \frac{t - j}{k - j}dt \right)$$

Abbiamo quindi ottenuto la formula che definisce l'approssimazione di $I(f)$ che stavamo cercando:

$$I_n(f) \equiv h \sum_{k=0}^{n} c_{kn} f_k$$

in cui

$$c_{kn} = \int_0^n \prod_{j=0, j\neq k}^{n} \frac{t - j}{k - j}dt, \quad k = 0, ..., n.$$

è detta formula di quadrtura di Newton-Cotes.

Teorema

Per i coefficienti c_{kn} vale che:

$$\frac{1}{n} \sum_{k=0}^{n} c_{kn} = 1$$

Dimostrazione:
considerando $f(x) \equiv 1$ si ha che:

$$\int_a^b 1dx = b - a \equiv \frac{b-a}{n} \sum_{k=0}^{n} c_{kn} = (b - a) \left(\frac{1}{n} \sum_{k=0}^{n} c_{kn} \right)$$

da cui deriva direttamente la tesi.

5.1.1 Formula dei trapezi

Poniamo $n = 1$. Sappiamo che $c_{01} + c_{11} = 1$, e quindi:

$$c_{11} = \int_0^1 \frac{t - 0}{1 - 0}dt = \frac{1}{2} \Longrightarrow c_{01} = 1 - c_{11} = \frac{1}{2}$$

e quindi:

$$I_1(f) = \frac{b-a}{2}(f(a) + f(b))$$

che approssima l'aria sottesa dal grafico di $f(x)$ con quella sottesa dal trapezio di vertici $(a, 0), (a, f(a)), (b, f(b))(b, 0)$.

5.1.2 Formula di Simpson

Se invece poniamo $n = 2$, sappiamo che $c_{02} + c_{12} + c_{22} = 2$, pertanto:

$$c_{22} = \int_0^2 \frac{t(t-1)}{2 \cdot 1}dt = \frac{1}{2} \int_0^2 (t^2 - t)dt = \frac{1}{2} \left(\frac{t^3}{3} - \frac{t^2}{2} \right) \Big|_0^2 = \frac{1}{3}$$

e

$$c_{02} = \int_0^2 \frac{(t-1)(t-2}{2 \cdot 1} dt = \frac{1}{2} \int_0^2 t(t-1) - 2(t-1) dt = \frac{1}{3} - \underbrace{\int_0^2 (t-1) dt}_{=0} = \frac{1}{3}$$

e si può ricavare $c_{12} = \frac{4}{3}$. Quindi:

$$I_2(f) = \frac{b-a}{6} \left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right)$$

Implementazione

Questo è il codice che implementa quanto appena descritto:

```
%Questo metodo applica la formula di quadratura di Newton-Cotes
%
%     I = NewtonCotes(f,a,b,n)
%
%Inputs:
%     f: la funzione integranda
%     a: estremo sinistro di integrazione
%     b: estremo destro di integrazione
%     n: grado (1 trapezi, 2 simpson...)
%
%outputs:
%
%     I: valore dell'integrale definito di f tra a e b
%

function I = NewtonCotes(f,a,b,n)

if n>4, error('Grado non impementato'), end
%Coefficienti:
%1 1/2 1/2
%2 1/3 4/3 1/3
%3 3/8 9/8 9/8 3/8
%4 14/45 64/45 24/45 64/45 14/45

cotes=[1/2,1/2,0,0,0,0,0;
       1/3,4/3,1/3,0,0,0,0;
       3/8,9/8,9/8,3/8,0,0,0;
       14/45,64/45,24/45,64/45,14/45,0,0;
       0,0,0,0,0,0,0;
       0,0,0,0,0,0,0];

h=(b-a)/n;

x=a:h:b;

F=feval(f,x);
I=0;
for i=1:n+1
    I=I+F(i)*cotes(n,i);
end
```

```
I=I*(b-a)/n;
```

5.1.3 Condizionamento del problema

Indichiamo con $\varphi(x)$ la perturbazione della funzione $f(x)$. Si ottiene:

$$|I_n(f) - I_n(\varphi)| = \frac{b-a}{n}\left|\sum_{k=0}^{n}(f_k - \varphi_k)c_{kn}\right| \leq \frac{b-a}{n}\sum_{k=0}^{n}|f_k - \varphi_k| \cdot |c_{kn}|$$

$$\leq \left(\frac{b-a}{n}\sum_{k=0}^{n}|c_{kn}|\right)\|f - \varphi\|$$

e quindi otteniamo che il numero di condizionamento del problema è:

$$k_n = \frac{b-a}{n}\sum_{k=0}^{n}|c_{kn}|$$

Osserviamo che :

$$k_n = k, \quad n = 1, ..., 6;$$
$$k_n > k, \quad n > 6.$$

Ne consegue quindi che i problemi di calcolo $I(f)$ e $I_n(f)$ hanno lo stesso numero di condizionamento solo per $n \leq 6$, e quindi le formule di Newton-Cotes sono sconvenienti al crescere di n.

5.2 Errore e formule composite

Esaminiamo adesso l'errore di quadratura, definito in questo modo:

$$E_n(f) = I(f) - I_n(f)$$

Dall'espressione dell'errore di interpolazione discende che:

$$E_n(f) = \int_a^b e(x)dx = \int_a^b (f(x) - p_n(x))\,dx = \int_a^b f[x_0, ..., x_n, x]\omega_{n+1}(x)dx.$$

Teorema

Se $f(x) \in \mathcal{C}^{(n+k)}$, con

$$k = \begin{cases} 1, & \text{se } n \text{ è dispari,} \\ 2, & \text{se } n \text{ è pari} \end{cases}$$

allora l'errore di quadratura è dato da:

$$E_n(f) = \nu_n \frac{f^{(n+k)(\xi)}}{(n+k)!}\left(\frac{b-a}{n}\right)^{n+k+1}$$

per un opportuno $\xi \in [a, b]$, ed:

$$\nu_n = \begin{cases} \int_0^n \prod_{j=0}^n (t-j)dt, & \text{se } n \text{ è dispari} \\ \int_0^n t\prod_{j=0}^n (t-j)dt, & \text{se } n \text{ è pari} \end{cases}$$

Per il metodo dei trapezi si ottiene quindi :

$$E_1(f) = -\frac{1}{12}f^{(2)}(\xi)(b-a)^3, \quad \xi \in [a, b]$$

e per quello di simpson:

$$E_2(f) = -\frac{1}{90} f^{(4)}(\xi) \left(\frac{b-a}{2}\right)^5, \quad \xi \in [a,b]$$

5.2.1 Formule composite

Per ridurre il rapporto $(b-a)/n$, è possibile pensare di suddividere l'intervallo $[a,b]$ in più sottointervalli della medesima ampiezza, ed utilizzare la stessa dormula di Newton-Cotes per ciascun sottointervallo.

Formula dei trapezi composita

Consideriamo l'applicazione della formula dei trapezi su ciascun sottointervallo $[x_{i-1}, x_i], i = 1, ..., n$ di una partizione uniforme di $[a,b]$:

$$\begin{aligned} I(f) &\approx I_1^{(n)}(f) \equiv \frac{b-a}{2n}(f_0 + f_1 + f_1 + f_2 + ... + f_{n-1} + f_{n-1} + f_n) \\ &= \frac{b-a}{2n}\left(f_0 + 2\sum_{i=1}^{n-1} f_i + f_n\right) \\ &= \frac{b-a}{n}\left((\sum_{i=0}^{n} f_i) - \frac{f_0 - f_n}{2}\right) \end{aligned}$$

Il cui errore di quadratura vale:

$$E_1^{(n)}(f) = -\frac{n}{12} f^{(2)}(\xi) \left(\frac{b-a}{n}\right)^3, \quad \xi \in [a,b].$$

Formula di Simpson composita

Considerando solo valori pari di n:

$$\begin{aligned} I(f) \approx I_2^{(n)}(f) &= -\frac{b-a}{3n}(f_0 + 4f_1 + f_2 + f_2 + ... + f_{n-2} + 4f_{n-1} + f_n) \\ &= \frac{b-a}{3n}\left(4\sum_{i=1}^{n/2} f_{2i-1} + 2\sum_{i=0}^{n/2} f_{2i} - f_0 - f_n\right) \end{aligned}$$

ed il corrispondente errore di quadratura:

$$E_2^{(n)}(f) = -\frac{n}{180} f^{(4)}(\xi) \left(\frac{b-a}{n}\right), \xi \in [a,b]$$

Notiamo come, per $n \to \infty$, $E_k^{(n)} \to 0$ con $k = 1, 2$.

Implementazione

Questo codice implementa la formula dei trapezi e quella di Simpson composite:

```
%Questo metodo applica le formula di quadratura dei trapezi oppure di Simpson
%in maniera composita.
%
%     I = NewtonCotesComp(f,a,b,n,m)
%
%Inputs:
%     f: la funzione integranda
%     a: estremo sinistro di integrazione
```

```
%     b: estremo destro di integrazione
%     n: grado
%     m: numeri di intervalli di composizione
%
%outputs:
%
%     I: valore dell'integrale definito di f tra a e b
%


% n grado (1 trapezi, 2 simpson)
% m numero intervalli

function I = NewtonCotesComp(f,a,b,n,m)

if n == 1
if (b-a) == 0, error('Intervallo troppo piccolo!'), end
x = [a:(b-a)/m:b];
F = feval(f,x);
I = 0;

for i = 2:m
I = I+F(i);
end

I = 2*I+F(1)+F(m+1);
I = I*((b-a)/(2*m));


elseif n==2
if mod(m,2)==1, error('Il numero di suddivisioni deve essere pari'), end
if (b-a) == 0, error('Intervallo troppo piccolo!'), end
x = [a:(b-a)/m:b];
F = feval(f,x);
I = 0;
S = 0;
        for i = 2:2:(m/2)+2
            I = F(i) + I;
        end
        for i = 1:2:(m/2)+3
            S = F(i) + S;
        end

I = 4*I+2*S-F(1)-F(m+1);
I = I*((b-a)/(3*m));
else
error('Grado non suopportato')
end
```

5.3 Formule adattative

Non è infrequente che la funzione integranda $f(x)$ abbia delle variazioni rapide solo in uno o più piccoli sottointervalli dell'intervallo $[a.b]$, e al contempo sia molto regolare

negli altri tratti. Quello che si vuole ottenere in questi casi è un metodo che si sappia adattare automaticamente a questo tipo di situazioni, ovvero che permetta di scegliere in modo opportuno i nodi della partizione dell'intervallo $[a, b]$.

5.3.1 Formula dei trapezi

applicando il metodo dei trapezi sull'intervallo $[a, b]$ diviso in 1 e 2 sottointervalli si ha:

$$I(f) \approx \begin{cases} I_1^{(1)}(f) = \dfrac{b-a}{2}(f(a) + f(b)) \\ I_1^{(2)}(f) = \dfrac{b-a}{4}(f(a) + 2f(\frac{a+b}{2}) + f(b)) \end{cases}$$

ovvero:

$$I(f) - I_1^{(1)}(f) \approx -\frac{1}{12} f''(\xi) \left(\frac{b-a}{1}\right)^3$$

$$I(f) - I_1^{(2)}(f) \approx -\frac{1}{12} f''(\xi) \frac{(b-a)^3}{4}$$

facendo la differenza membro a membro si ottiene:

$$I_1^{(2)}(f) - I_1^{(1)}(f) \approx -\frac{1}{12} f''(\xi)(b-a)^3 \left(\frac{1}{4} - 1\right)$$

e ancora dividendo per -3:

$$\frac{-I_1^{(2)}(f) + I_1(f)}{3} \approx -\frac{1}{12} f^{(2)}(\xi)(b-a)^3 \frac{1}{4} \approx I(f) - I_1^{(2)}(f).$$

Abbiamo appena trovato una stima dell'errore:

$$I(f) - I_1^{(2)}(f) \approx \frac{1}{3}(I_1^{(2)}(f) - I_1^{(1)}(f))$$

che possiamo utilizzare come soglia in un algoritmo ricorsivo per decidere se è il caso di suddividere ulteriormente il sottointervallo considerato.

5.3.2 Formula di Simpson

applicando il metodo di Simpson sull'intervallo $[a, b]$ diviso in 2 e 4 sottointervalli si ha:

$$I(f) \approx \begin{cases} I_2^{(2)}(f) = \dfrac{b-a}{6}(f(a) + 4f(\frac{a+b}{2}) + f(b)) \\ \\ I_2^{(4)}(f) = \dfrac{b-a}{12}(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + f(x_4)) \end{cases}$$

dove $x_0 = a, x_2 = \frac{a+b}{2}, x_4 = b, x_1 = \frac{x_0+x_2}{2}, x_3 = \frac{x_2+x_4}{2}$.
Si ottiene quindi:

$$I_1(f) - I_2^{(2)}(f) \approx -\frac{1}{180} f^{(4)}(\xi) \left(\frac{b-a}{2}\right)^5$$

$$I_1(f) - I_2^{(4)}(f) \approx -\frac{1}{180} f^{(4)}(\xi) \left(\frac{(b-a)^5}{2^5 \cdot 16}\right)$$

da cui:

$$I_2^{(4)}(f) - I_2^{(2)}(f) \approx -\frac{1}{180} f^{(4)}(\xi) \frac{(b-a)^5}{2^5}(1 - 16) = -\frac{1}{180} f^{(4)}(\xi) \frac{(b-a)^5}{2^5 \cdot 16}(16 - 1)$$

Abbiamo appena trovato una stima dell'errore:

$$I(f) - I_2^{(4)}(f) \approx \frac{I_2^{(4)} - I_2^{(2)}}{15}$$

che come nel caso della formula dei trapezi possiamo utilizzare come soglia per decidere se è il caso di suddividere ulteriormente il sottointervallo considerato.

Implementazione

Questi codici implementano in maniera ricorsiva e molto semplice le formule adattive dei trapezi e di Simpson:

```
%formula adattiva dei Trapezi
%
%     [I,p] = SimpsonAdattivaNRfop(f,a,b,tol)
%
%Questo metodo prende in input
%     f: funzione integranda
%     a: estremo sinistro dell'intervallo di integrazione
%     b: estremo destro dell'intervallo di integrazione
%     tol: tolleranza sul risultato
%e restituisce:
%     I: valore dell'integrale
function I = TrapeziAdattiva(f,a,b,tol)

if a==b I =0; return, end



I1 =  NewtonCotesComp(f,a,b,1,1);

I2 =  NewtonCotesComp(f,a,b,1,2);


if (abs(I1-I2)/3)<=tol I = I2; return, end

I = TrapeziAdattiva(f,a,(a+b)/2,tol/2)+TrapeziAdattiva(f,(a+b)/2,b,tol/2);

%formula adattiva di Simpson
%
%     [I,p] = SimpsonAdattivaNRfop(f,a,b,tol)
%
%Questo metodo prende in input
%     f: funzione integranda
%     a: estremo sinistro dell'intervallo di integrazione
%     b: estremo destro dell'intervallo di integrazione
%     tol: tolleranza sul risultato
%e restituisce:
%     I: valore dell'integrale

function I = SimpsonAdattiva(f,a,b,tol)

if a==b I =0; return, end
```

```
I1 = NewtonCotes(f,a,b,2);
I2 = NewtonCotes(f,a,(a+b)/2,2)+NewtonCotes(f,(a+b)/2,b,2);


if abs(I1-I2)/15<=tol I = I2; return, end

I = SimpsonAdattiva(f,a,(a+b)/2,tol/2)+SimpsonAdattiva(f,(a+b)/2,b,tol/2);
```

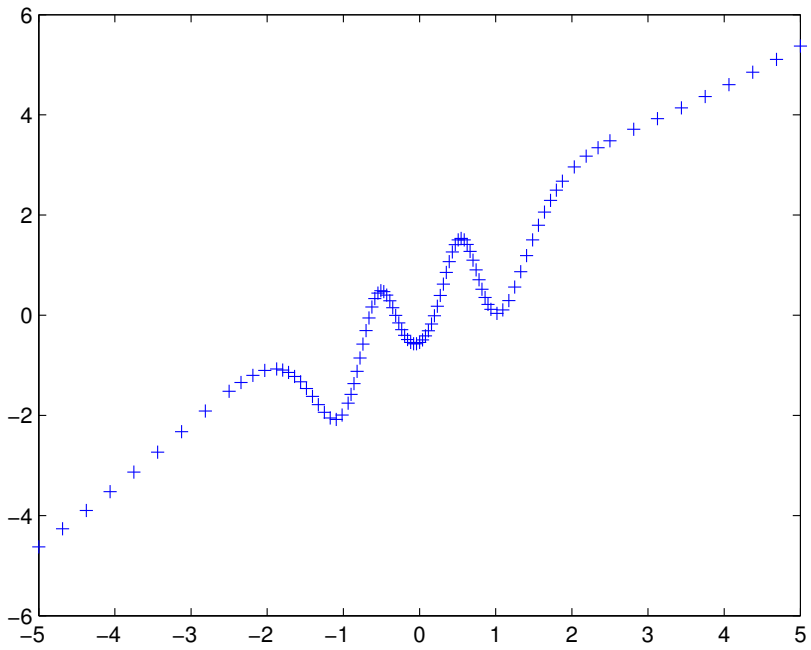

Figura 5.1: Questo grafico mostra il metodo di simpson adattativo sulla funzione $sin(10/(1+x^2))+x$ con tolleranza 10^{-3}.

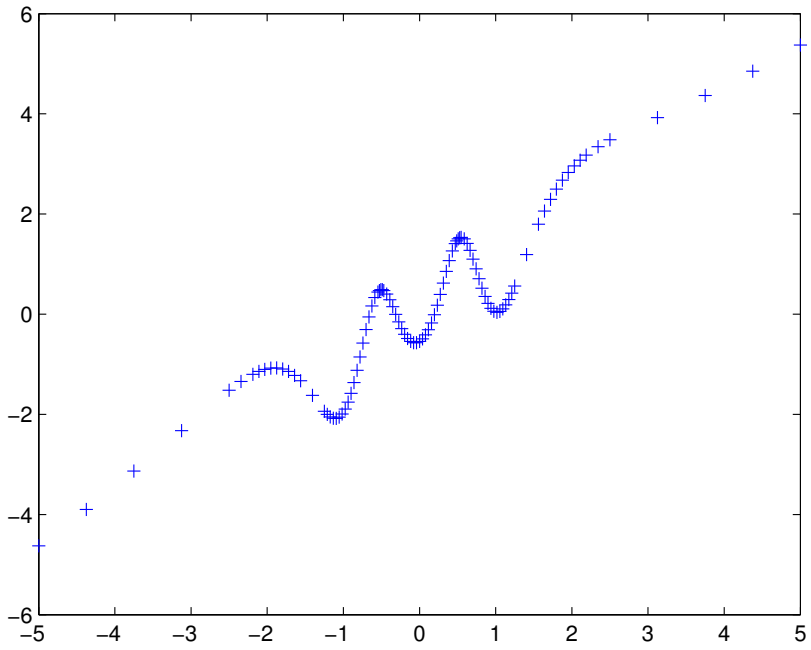

Figura 5.2: Ecco il comportamento sulla stessa funzione del metodo adattivo dei trapezi, ma con tolleranza $5 \cdot 10^{-2}$. A parità di tolleranza il metodo dei trapezi lavora su 697 punti contro solo 97 del metodo di Simpson del grafico sopra.

Capitolo 6

Altre implementazioni

6.1 Modifica al metodo delle secanti

Questo metodo, invece di richiedere la formula della derivata prima come fa il metodo originale, calcola un'approssimazione del valore di questa nel punto iniziale x_0:

```
%Questo metodo determina uno zero della funzione in ingresso utilizzando il
%metodo delle secanti.
%
% [x,i] = secantimod(f,x0,itmax,tolx,rtolx)
%
%Questo metodo prende in input:
%  f: la funzione di cui si vuol trovare uno zero
%  imax: numero massimo di iterazioni consentite
%  tolx: tolleranza assoluta sul valore dello zero
%  rtolx: tolleranza relativa sul valore dello zero
%
%Questo metodo restituisce:
%  x: zero della funzione
%  i: numero di iterazioni fatte
%

function [x,i] = secantimod(f,x0,itmax,tolx,rtolx)
   i=0;
   fx = feval(f,x0);
%Invece di richiedere la derivata della funzione, ne calcoliamo una
%apprissimazione.
   h = 1e-10;
   f1x = (feval(f,(x0+h))-fx)/h;
   fxx(1)=fx;
   xx(1) = x0;

   if f1x == 0
       if fx==0
           return
       end
       error('Derivata prima nulla, impossibile continuare.')
   end
```

```
x = x0 - fx/f1x;

while (i<itmax) & ((abs(x-x0)/(tolx+rtolx*(abs(x))))>1)
    i = i+1;
    fx0 = fx;
    fx = feval(f,x);
    t = (fx-fx0);
    if t == 0
      if fx == 0
          return
      elseif ((abs(x-x0)/(tolx+rtolx*(abs(x))))<=1)
         return
      end
     error('Impossibile determinare la radice nella tolleranza desiderata')
 end
    x1 = (fx*x0-fx0*x)/t;
    x0 = x;
    x = x1;
end
if ((abs(x-x0)/(tolx+rtolx*(abs(x))))>1), disp('Il metodo non converge'), end
```

In questa tabella confrontiamo i valori ottenuti con i due metodi:

Cercando uno zero della funzione $f(x) = x^3 + sin(10x)$:

x_0	tolx	iter.S	ris. Secanti	iter.SM	ris. Sec. Mod.	errore
0.2	10^{-2}	2	0.31674403271112	2	0.31674403804563	5e-09
0.2	10^{-6}	4	0.31735605718908	4	0.31735605718908	1e-15
0.2	10^{-10}	5	0.31735605713204	5	0.31735605713204	5e-17
1.5	10^{-2}	15	-0.31053322118703	17	-0.31735615577029	6e-3
1.5	10^{-4}	18	-0.31735605713204	18	-0.31735605713245	4e-13
1.5	10^{-10}	19	-0.31735605713204	19	-0.31735605713204	5e-17
1.06	10^{-2}	47	-8.549970108032228e-06	25	0.31666506455617	-
1.06	10^{-4}	49	-8.844156346925046e-21	27	0.31735605728114	-
1.06	10^{-10}	49	-8.844156346925046e-21	29	0.31735605713204	-

Vediamo quindi che la valutazione approssimata della derivata porta in un caso a convergere verso una radice diversa rispetto a quella cui converge il metodo originale (ma comunque è un risultato corretto), e negli altri casi non peggiora il risultato.

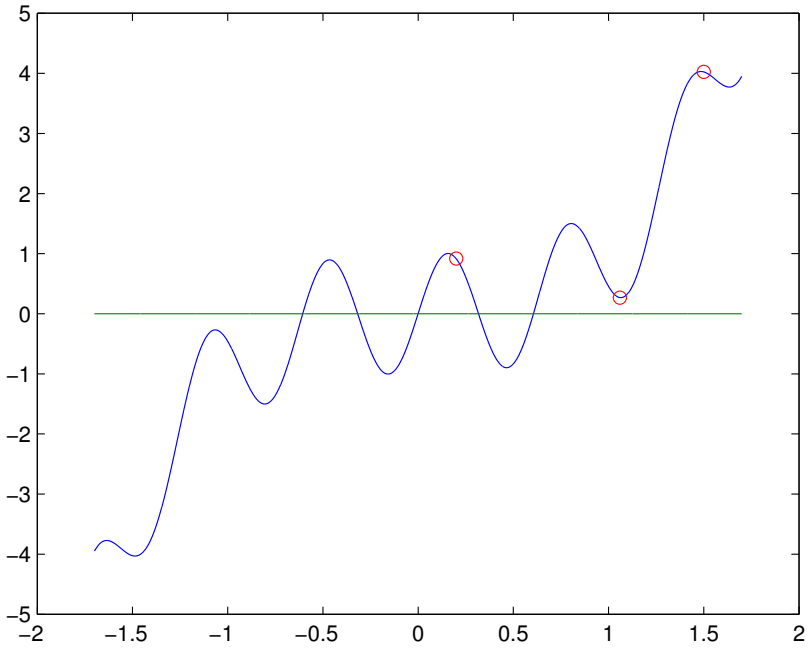

Figura 6.1: Questo grafico mostra la funzione $f(x) = x^3 + sin(10x)$, i cerchi rossi indicano gli x_0 utilizzati come punti di partenza dei metodi.

6.2 Fattorizzazione LU

6.2.1 Algoritmo ottimizzato

Questa è la versione ottimizzata dell'algoritmo presentato nella parte teorica. Con queste semplici modifiche si riesce ad ottenere un aumento di prestazioni notevole: fattorizzando una matrice di elementi random di 2000x2000 elementi, si passa dai 195.73 secondi dell'algoritmo originale a 74.32.

```
%Fattorizzazione LU
%
% [A]=fattLU(A)
%
%La funzione prende in input
% A: una matrice quadrata
%E restituisce
% A: una matrice contenente la porzione strettamente triangolare
% inferiore della matrice L e la porzione triangolare superiore
% della matrice U
% t: tempo di computazione

function [A]=fattLU(A)

[m,n]=size(A);
if m~=n, error('La matrice non e quadrata'), end

for i = 1:n-1
```

```
    if A(i,i)==0
        error('Matrice non fattorizzabile LU')
    end
    for k = i+1:n
A(k,i) = A(k,i)/A(i,i);
    end


    for c = i+1:n
        k = A(i,c);
        for r = i+1:n
            A(r,c) = A(r,c)-k*A(r,i);
        end
    end

end
```

6.2.2 Ottenere i fattori L ed U

Con questo algoritmo si recuperano dalla matrice A restituita dall'algoritmo di fattorizzazione i fattori L ed U necessari alla effettiva risoluzione del sistema:

```
%Questa funzione separa i fattori L ed U nella matrice risultante dalla
%fattorizzazione LU della funzione fattLU.
%
% [L,U] = unFattLU(A)
%
%Questo metodo prende input:
% A: matrice contenente la fattorizzazione LU
%   con la parte strettamente triangolare inferiore di L
%   e la parte triangolare superiore di U
%E restituisce:
% L: matrice L della fattorizzazione LU
% U: matrice U della fattorizzazione LU
%

function [L,U] = unFattLU(A)
n = length(A);
for i = 1:n
    L(i,i)=1;
end
for j = 1:n
    L(j+1:n,j) = A(j+1:n,j);
    U(1:j,j) = A(1:j,j);
end
```

6.2.3 Risolvere il sistema

Questo semplice codice automatizza la risoluzione del sistema $Ax = b$:

```
%Questo metodo risolve un sistema lineare del tipo Ax=b mediante
%fattorizzazione LU.
```

```
%
% [b] = solveLU(A,b)
%
%Questo metodo prende in input:
% A: matrice quadrata
% b: vettore dei termini noti
%E restituisce
% b: vettore delle soluzioni del sistema
%
function [b,t] = solveLU(A,b)

if size(A,1)~=size(b,1)
error('Le dimensione di A e di b non sono compatibili')
end

[A,U] = unFattLU(fattLU(A));

b = sisem(U,sisem(A,b,'inf'),'sup');
```

6.2.4 Ottimizzazione per sistemi tridiagonali

In caso si debba fattorizzare un sistema tridiagonale si può migliorare l'algoritmo:

```
%Fattorizzazione LU di una matrice A tridiagonale
%
% [A]=fattLUtridiag(A)
%
%La funzione prende in input
% A: una matrice quadrata
%E restituisce
% A: una matrice contenente la porzione strettamente triangolare
%    inferiore della matrice L e la porzione triangolare superiore della
%    matrice U
% t: tempo di computazione

function [A]=fattLUtridiag(A)

[m,n]=size(A);
if m~=n, error('La matrice non e quadrata'), end

for i = 1:n-1

    if A(i,i)==0
        error('Matrice non fattorizzabile LU')
    end

    A(i+1:n,i) = A(i+1:n,i)/A(i,i);

    for c = i+1:n
        k = A(i,c);
        for r = i+1:c
            A(r,c) = A(r,c)-k*A(r,i);
        end
```

```
    end
 %   A(i+1:n,i+1:n) = A(i+1:n,i+1:n)-A(i+1:n,i)*A(i,i+1:n);
end
```

6.3 Fattorizzazione LDL^T

6.3.1 Algoritmo ottimizzato

Questa versione ottimizzata dell'algoritmo presentato nella parte teorica utilizza un vettore contenente la porzione triangolare inferiore della matrice A anziché tutta la matrice, sfruttandone la simmetria. Il codice può essere ulteriormente ottimizzato integrando le funzionalità dei metodi vget e vstore nel codice; viene effettuata una chiamata al metodo per ogni accesso ad un elemento della matrice, e questo appesantisce parecchio dal punto di vista computazionale (ma facilita la lettura del codie).

```
%Fattorizzaziole LDL^T
%
% [v] = fattLDLold(v,n)
%
%Questo metodo prende in input:
% v: un vettore contente la porzione triangolare inferiore di
%    una matrice sdp per colonne
% n: la dimensione della matrice
%
%E restituisce:
% v: un vettore contenente la porzione triangolare inferiore di una
%    matrice contente L nella porzione strettamente triangolare
%    e D sulla diagonale
% t: il tempo di computazione

function v = fattLDLold(v,n)
%if length(v) ~= (n^2+n)/2
% error('Il vettore non rappresenta una matrice n x n')
%end


    if v(vget(1,1,n)) <= 0, error('La matrice non e'' sdp'), end

    %A(2:n,1) = A(2:n,1)/A(1,1);
    k = v(vget(1,1,n));
    for i = 2:n
       v(vget(i,1,n)) = v(vget(i,1,n))/k;
    end

    for j=2:n


        %v = ( A(j,1:j-1)').*diag(A(1:j-1,1:j-1)); %ljk * dk
        for i = 1:j-1
             vt(i) = v(vget(j,i,n))*v(vget(i,i,n));
        end

        %A(j,j) = A(j,j)-A(j,1:j-1)*v; %v * ljk
```

```
        x=0;
        for i = 1:j-1
           x  = v(vget(j,i,n))*vt(i)+x;
        end
         v(vget(j,j,n)) = v(vget(j,j,n))-x;
        if x<=0, error('La matrice non e'' fattorizzabile LDL'), end

        %A(j+1:n,j)= (A(j+1:n,j)-A(j+1:n,1:j-1)*v)/A(j,j);
for i = j+1:n
k=0;
sum = 0;
for k = 1:j-1
sum = sum + v(vget(i,k,n))*vt(k);
end
x = v(vget(i,j,n)) - sum;
v(vget(i,j,n)) = x/v(vget(j,j,n));
end

     end


%matrice memorizzata per colonne

%calcolo dell'indice del vettore a partire dalla posizione
%     richiesta della matrice
function k = vget(i,j,n)
        t = 0;
if i<j
t = i ;
i = j;
j = t;
end
s = 0;
for z =0:j-1
s = s + (n - z);
end
k = s - (n-i) ;
```

6.3.2 Trasformare una matrice sdp in un vettore utile all'utilizzo dell'algoritmo ottimizzato

Questo codice serve per vettorizzare per righe la porzione triangolare inferiore di una matrice:

```
%Da matrice sdp a vettore per colonne

function v = vectorizeLDL(A)

 [m,n] = size(A);

  if m~=n, error('Matrice non quadrata'), end

k = 0;
  for j = 1:n
```

```
   for i = j:n
k = k+1;
v(k) = A(i,j);
end
end
```

Questa è un'ottimizzazione del codice di vettorizzazione:

```
%Da matrice sdp a vettore per colonne

function v = vectorizeLDLopt(A)

 [m,n] = size(A);

  if m~=n, error('Matrice non quadrata'), end

k = 0;
v = 0;
v(1:n) = A(1:n,1)';
for j = 2:n
        l = length(v);
   v(l+1:(l+(n-j))+1) = A(j:n,j)';
end
end
```

6.3.3 Piccolo miglioramento

Questa versione del metodo elimina le chimate di funzione per il calcolo degli indici:

```
%Fattorizzaziole LDL^T
%
% [v] = fattLDL(v,n)
%
%Questo metodo prende in input:
% v: un vettore contente la porzione triangolare inferiore di
%    una matrice sdp per colonne
% n: la dimensione della matrice
%
%E restituisce:
% v: un vettore contenente la porzione triangolare inferiore di una
%    matrice contente L nella porzione strettamente triangolare
%    e D sulla diagonale

function v = fattLDL(v,n)
%if length(v) ~= (n^2+n)/2
% error('Il vettore non rappresenta una matrice n x n')
%end



if v(1) <= 0, error('La matrice non e'' sdp'), end

%A(2:n,1) = A(2:n,1)/A(1,1);
```

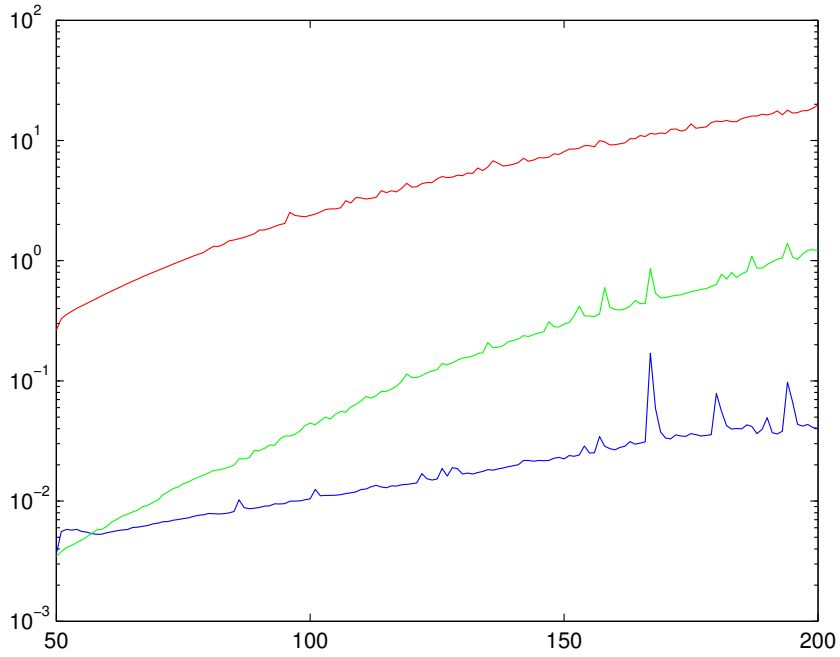

Figura 6.2: Questo grafico mostra il costo medio temporale per la fattorizzazione LDL di matrici di dimensioni variabili da 50×50 a 200×200 elementi. In Blu è rappresentato il tempo impiegato dalla fattorizzazione con matrice, in rosso la fattorizzazione LDL vettorizzata, in verde il tempo impiegato per trasformare la matrice in vettore. Notiamo come la vettorizzazione, benchè permetta un grosso risparmio in termini di memoria (memorizziamo solo circa la metà degli elementi della matrice) aumenta non di poco il tempo di computazione, pagato in termini di trasferimenti di porzioni di memoria e chiamate a funzioni esterne.

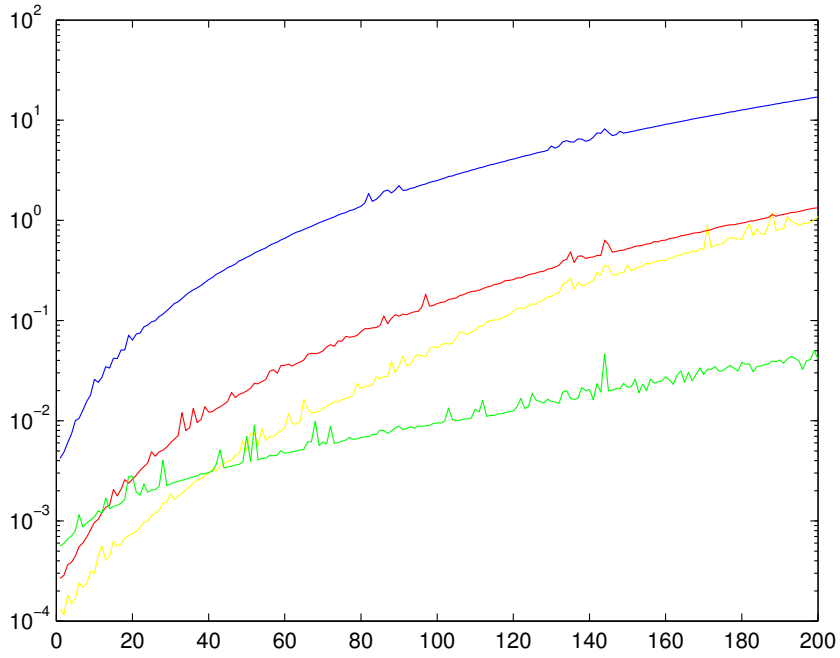

Figura 6.3: Questo grafico confronta il costo temporale all'aumentare della dimensione della matrice dell'algoritmo di fattorizzazione con le chiamate di funzione (blu), quello senza chiamate (rosso) e quello originale (verde). Notiamo come le chiamate di funzione appesantissero in maniera notevole il metodo. Il giallo indica il tempo di vettorizzazione (non ottimizzato).

```
k = v(1);
for i = 2:n
v(i) = v(i)/k;
end

    for j=2:n


        %v = ( A(j,1:j-1)').*diag(A(1:j-1,1:j-1)); %ljk * dk
for i = 1:j-1

index = 0;
for z =0:i-1
index = index + (n - z);
end
index = index - ( n - j ) ;

index1 = 0;
for z =0:i-1
index1 = index1 + (n - z);
end
index1 = index1 - n + i;

vt(i) = v(index)*v(index1);
end

        %A(j,j) = A(j,j)-A(j,1:j-1)*v; %v * ljk
x=0;
for i = 1:j-1

index = 0;
for z =0:i-1
index = index + (n - z);
end
index = index - ( n - j ) ;

x  = v(index)*vt(i)+x;
end

index1 = 0;
for z =0:j-1
index1 = index1 + (n - z);
end
index1 = index1 - n + j;

v(index1) = v(index1)-x;



if x<=0, error('La matrice non e'' fattorizzabile LDL'), end

        %A(j+1:n,j)= (A(j+1:n,j)-A(j+1:n,1:j-1)*v)/A(j,j);
for i = j+1:n
k=0;
```

```
sum = 0;
for k = 1:j-1

index = 0;
for z =0:k-1
index = index + (n - z);
end
index = index - ( n - i ) ;

sum = sum + v(index)*vt(k);
end

index1 = 0;
for z =0:j-1
index1 = index1 + (n - z);
end
index1 = index1 - ( n - i ) ;

x = v(index1) - sum;

index = 0;
for z =0:j-1
index = index + (n - z);
end
index = index - ( n - i ) ;


index1 = 0;
for z =0:j-1
index1 = index1 + (n - z);
end
index1 = index1 - n + j;

v(index) = x/v(index1);
end

     end
```

6.3.4 Ottenere i fattori L e D

Con questo algoritmo si recuperano dalla matrice A restituita dall'algoritmo di fattorizzazione non ottimizzato i fattori L e D necessari alla effettiva risoluzione del sistema:

```
% Estrapola i fattori L e D a partire dalla matrice che li contiene
%
% [L,D]= unfattLDLlib(A)
%

function [L,D]= unfattLDLlib(A)
n = length(A);
```

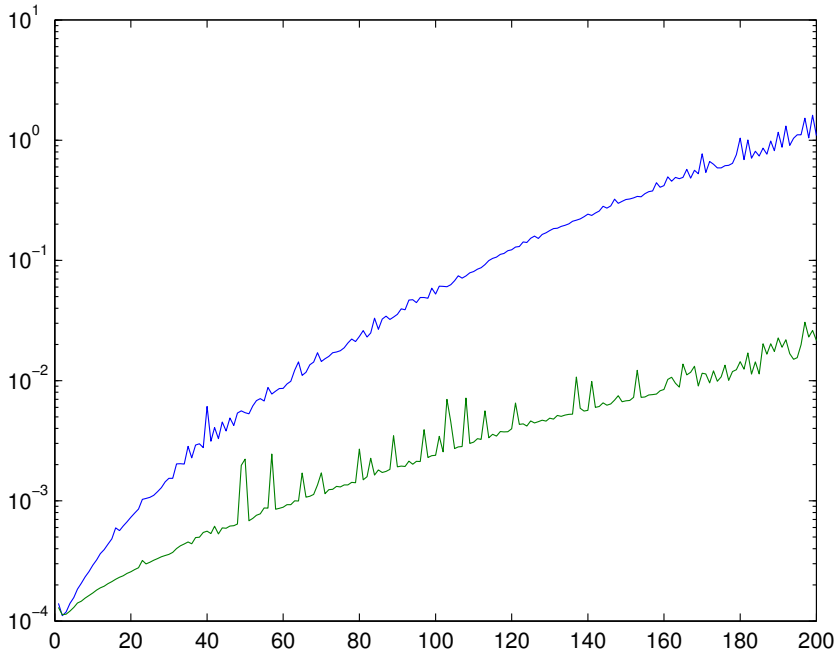

Figura 6.4: Questo grafico confronta il costo temporale all'aumentare della dimensione della matrice dell'algoritmo di vettorizzazione non ottimizzato (blu) con quello ottimizzato (verde). Questo confronto mostra come l'efficienza aumenti lavorando su blocchi di dati anzichè dati singoli.

```
%L ha diagonale unitaria
for i = 1:n
    L(i,i)=1;
end

for j = 1:n
    L(j+1:n,j) = A(j+1:n,j);
    D(j,j) = A(j,j);
end
```

E con questo si fa la stessa cosa a partire dal vettore restituito dalla versione vettorizzata:

```
%Estrapola i fattori L e D a partire da un vettore che contiene la porzione
%triangolare inferiore della matrice A per colonne
%
% [L,D]= unfattLDL(v,n)
%
  function [L,D]= unfattLDL(v,n)

for j = 1:n
    for i = j:n
        A(i,j)=v(vget(i,j,n));
    end
end

%L ha diagonale unitaria
for i = 1:n
    L(i,i)=1;
end

for j = 1:n
    L(j+1:n,j) = A(j+1:n,j);
    D(j,j) = A(j,j);
end

%Funzione che estrapola l'elemento in posizione (i,j) dal vettore
function k = vget(i,j,n)

if i<j
t = i ;
i = j;
j = t;
end

s = 0;

for z =0:j-1
s = s + (n - z);
end

k = s - (n-i) ;
```

6.3.5 Risolvere il sistema

Questo semplice codice automatizza la risoluzione del sistema $Ax = b$:

```
%Questo metodo risolve un sistema lineare del tipo Ax=b mediante
%fattorizzazione LDL'.
%
% [b,t] = solveLDL(A,b,d)
%
%Questo metodo prende in input:
% A: matrice sdp oppure vettore contenente la porzione triangolare
%    inferiore della matrice per righe
% b: vettore dei termini noti
% d: dimensione della matrice in caso si utilizzi il vettore
%
%E restituisce
% b: vettore delle soluzioni del sistema
%
function [b,t] = solveLDL(A,b,d);
tic;

[m,n] = size(A);
if size(A,1)~=size(b,1)
error('Le dimensione di A e di b non sono compatibili')
end

%Caso del vettore
if min(n,m)==1
if d>0
A = fattLDL(A,d)
[L,D]= unfattLDL(A,d);
b = sisem(L',sisem(L*D,b,'inf'),'sup');
else
error('Dati incorretti');
end

else
%Caso della matrice

A = fattLDLlib(A);
[L,D]= unfattLDLlib(A);
U = D*L';
b = sisem(L',sisem(L*D,b,'inf'),'sup');

end
t = toc;
```

6.4 Fattorizzazione LU con pivoting parziale

6.4.1 Algoritmo ottimizzato

Come nel caso della fattorizzazione LU, svolgendo a mano i cicli che Matlab svolge in maniera inefficiente guadagnamo del tempo:

```
%Fattorizzazione con pivoting parziale
```

```
%
% [A,p] = fattPivot(A)
%
%Questo metodo prende in input:
% A:  una matrice
%E restituisce:
% A: la matrice contenente la fattorizzazione
% p: il vettore di permutazione
% t: il tempo di computazione

function [A,p] = fattPivot(A)

n= length(A);
p = [1:n];
for i = 1:n-1
    [mi,ki] = max(abs(A(i:n,i))); %mi elemento massimo,
  %ki indice riga relativo sottomatrice

    if mi==0, error('Matrice singolare'), end
    ki = ki +i-1;

    if ki > i
        A([i ki],:) = A([ki i],:);
        p([i ki]) = p([ki i]);
    end

    A(i+1:n,i) = A(i+1:n,i)/A(i,i);

    for c = i+1:n
        k = A(i,c);
        for r = i+1:n
            A(r,c) = A(r,c)-k*A(r,i);
        end
   end
end
```

6.4.2 Risolvere il sistema

Questo codice automatizza la risoluzione del sistema $Ax = b$:

```
%Questo metodo risolve un sistema lineare del tipo Ax=b mediante
%fattorizzazione LU con pivoting parziale.
%
% [b] = solveLUpivot(A,b)
%
%Questo metodo prende in input:
% A: matrice quadrata
% b: vettore dei termini noti
%E restituisce
% b: vettore delle soluzioni del sistema
%
function [b] = solveLUpivot(A,b)

if size(A,1)~=size(b,1)
```

```
error('Le dimensione di A e di b non sono compatibili')
end
[A,p,t] = fattPivot(A);
b = permuta(b,p);
[A,U] = unFattLU(A);
b = sisem(U,sisem(A,b,'inf'),'sup');
```

in cui utilizziamo il metodo di permutazione:

```
%Permuta una matrice a partire da un vettore di permutazione
%
%  A = permuta(A,p)
%
%Questo metodo prende in input:
% A: matrice da permutare
% p: vettore di permutazione tale che length(p)=size(A)
%E restituisce:
% A: matrice permutata

function A = permuta(A,p)
n = length(p);
if size(A,1) ~= n, error('Matrice e vettore non compatibili'), end
s = zeros(n,1);
for i = 1:n-1
   if (i<p(i)), A([i p(i)],:) = A([p(i) i],:);, s(i)=s(i)+1;,  end
   if i>p(i)
       k = i;
       c = 1;
       while(c~=0)
        k=p(k);
        c = s(k);
       end
       A([k i],:) = A([i k],:);,s(i)=s(i)+1;
   end
end
```

6.5 Fattorizzazione QR

6.5.1 Ottenere i fattori Q' ed R

Con questo algoritmo recuperiamo i fattori Q' ed R direttamente utili alla risoluzione del sistema; l'algoritmo di fattorizzazione restituisce nella porzione triangolare inferiore della matrice i vettori di Householder ed è pertanto necessario calcolare la matrice Q', R è facilmente costruibile a partire da $\hat{R}$.

```
%Questa funzione separa i fattori Q' ed R dalla matrice A risultante dalla
%fattorizzazione QR restituita dalla funzione fattQRhouseholder.
%
% [Q,R] = unFattQR(A)
%

function [Q,R] = unFattQR(A)
[m,n] = size(A);
```

```
%Costruisce R a partire da \hat{R}
for j= 1:n
    for i = 1:m
        if i>n
            R(i,j)=0;
        elseif j>=i
            R(i,j)=A(i,j);
        end
    end
end

Q = eye(m);

%Calcola Q' a partire dai vettori di Householder
for k = 1:n
    v = ones(m-k,1);
    v(2:m-k+1) = A(k+1:m,k);
    Qk = eye(m);
    Qk(k:m,k:m) = Qk(k:m,k:m) - (2/(v'*v)*(v*v'));
    Q = Qk*Q;
end
```

6.5.2 Risolvere il sistema

Anche in questo caso abbiamo un semplice codice per automatizzare la risoluzione del sistema $Ax = b$:

```
%Questo metodo risolve un sistema lineare del tipo Ax=b mediante
%fattorizzazione QR con metodo di Householder.
%
% [b] = solveQR(A,b)
%
%Questo metodo prende in input:
% A: matrice rettangolare con m>n
% b: vettore dei termini noti di dimensione m.
%E restituisce
% b: vettore delle soluzioni del sistema
%
function b = solveQR(A,b)
if size(A,1)~=size(b,1)
error('Le dimensione di A e di b non sono compatibili')
end
A = fattQRhouseholder(A);
[Q,R] = unFattQR(A);
[n,m]=size(R);
g = Q*b;
b = sisem(R(1:m,:),g(1:m),'sup');
```

6.6 Algoritmo di Horner

6.6.1 Algoritmo originale

Questo algoritmo valuta il polinomio $p(x) = \sum_{k=0}^{n} a_k x^k$ nei punti contenuti nel vettore x a partire dal vettore a contenente i coefficienti del polinomio. Quello che si fa infatti è svolgere il calcolo: $a_1 + x(a_2 + x(a_3 + x(...a_n x)))$ che svolgendo i calcoli si vede essere equivalente a: $a_1 + xa_2 + x^2 a_3 + ... + x^n a_n$.

```
%algoritmo di horner
%a contiene i coefficienti da quello di grado piu alto a quello di grado 1.

function p = Horner(a,x)

n = length(a);
lx = length(x);

p(1,1:lx) = a(1);

for i = 2:n
    p = p.*x+a(i);
end
```

6.6.2 Algoritmo greneralizzato

Questo algoritmo valuta il polinomio $p(x) = a_1 + (x - x_1)a_2 + (x - x_1)(x - x_2)a_3 + ... + (x - x_1)...(x - x_{n-1})a_n$ nei punti contenuti nel vettore x_0 a partire dal vettore a contenente i coefficienti del polinomio e il vettore x contenente le radici del polinomio.

```
%Algoritmo di horner generalizzato
%Questo metodo prende in input:
%   f : array dei coefficienti
%   x : array degli zeri del polinomio
%   x0: array di punti di valutazione del polinomio
% e restituisce la valutazione del polinomio
% pr-1(x)+fwr(x) nei punti x0

function p = HornerGen(x,f,x0)

n = length(f);
if n~=length(x), error('Vettori f e x di lunghezza diversa'), end

    p = f(n);
    for i = n-1:-1:1
        p = p.*(x0-x(i))+f(i);
    end
```

6.6.3 Polinomio interpolante di Newton

```
%Calcola il polinomio interpolante di Newton
%
%  ff = ValutaPolNewton(x,f,x0)
%
%Questo metodo prende in input:
```

```
% x: vettore delle ascisse di interpolazione
% f: vettore dei valori della funzione su x
% x0: vettore dei punti incui valutare il polinomio
%
%e restituisce:
% ff: vettore dei valori del polinomio valutato sui punti x0.
%
function ff = ValutaPolNewton(x,f,x0)

dd = NewtonDiffDiv(x,f);

ff = HornerGen(x,dd,x0);
```

6.6.4 Polinomio interpolante di Hermite

```
%Calcola il polinomio interpolante di Hermite
%
%  ff = ValutaPolHermite(x,f,x0)
%
%Questo metodo prende in input:
% x: vettore delle ascisse di interpolazione
% f: vettore dei valori della funzione su x
% x0: vettore dei punti incui valutare il polinomio
%
%e restituisce:
% ff: vettore dei valori del polinomio valutato sui punti x0.
%
function ff = ValutaPolHermite(x,f,f1,x0)
n=length(x);

xx(1:2:2*n)=x(1:n);
xx(2:2:2*n)=xx(1:2:2*n);

ff(1:2:2*n)=f(1:n);
ff(2:2:2*n)=f1(1:n);

dd = HermiteDiffDiv(xx,ff);

ff = HornerGen(xx,dd,x0);
```

6.7 Spline

6.7.1 Spline lineare

Questo semplice codice calcola e valuta una spline lineare:

```
%Questo metodo calcola e valuta una spline lineare
%
% [x0] = ValutaSplineLineare(x,f,x0)
%
%Inputs:
% x: ascisse di interpolazione
% f: valori della funzione corrispondenti alle ascisse di interpolazione
% x0: vettore dei punti incui calcolare la spline
```

```
%
%Outputs:
% x0: vettore contenete il valore della spline sui punti
%     presi in input
%

function [x0] = ValutaSplineLineare(x,f,x0)

n = length(x);
k = length(x0);

for c = 1:k
    for i = 2:n
        if (x0(c) <= x(i)) & (x0(c) >= x(i-1))
            x0(c)=(((x0(c)-x(i-1))*f(i))+(x(i)-x0(c))*f(i-1))/(x(i)-x(i-1));
        end
    end
end
```

6.7.2 Spline cubica naturale

Questi codici effettuano il calcolo e la valutazione in un array di punti di una spline cubica naturale:

```
%Questo metodo calcola e valuta una spline cubica naturale
%
% x0 = ValutaSplineCubicaNaturale(x,y,x0)
%
%Inputs:
% x: ascisse di interpolazione
% y: valori della funzione corrispondenti alle ascisse di interpolazione
% x0: vettore dei punti incui calcolare la spline
%
%Outputs:
% x0: vettore contenete il valore della spline sui punti
%     presi in input
%

function x0 = ValutaSplineCubicaNaturale(x,y,x0)


[m,r,q,h] = CalcolaSplineCubicaNaturale(x,y);

k = length(x0);
n = length(x);

for c = 1:k

      if (( (x0(c)) <= (x(2))) & ((x0(c)) >= (x(1))))

            x0(c) = q(1)*(x0(c)-x(1))+r(1)+((x0(c)-x(1))^3*m(1))/(6*h(1));

      elseif (((x0(c)) <= (x(n))) & ((x0(c)) >= (x(n-1))))
```

```
            x0(c) = r(n-1) + q(n-1)*(x0(c)-x(n-1)) +...
                                ( (((x(n)-x0(c))^3)*(m(n-2)))/(6*h(n-1)));

      else

      for i = 2:n-2
            if (((x0(c)) <= (x(i+1))) & ((x0(c)) >= (x(i))))
            x0(c) = r(i)+q(i)*(x0(c)-x(i))+( ((x0(c)-x(i))^3*m(i)...
                                + (x(i+1)-x0(c))^3*m(i-1))/(6*h(i))  );

            break
            end

      end
      end


end

%Questo metodo calcola  una spline cubica naturale
%
% [m,r,q,h] = CalcolaSplineCubicaNaturale(x,f)
%
%Inputs:
% x: ascisse di interpolazione
% f: valori della funzione corrispondenti alle ascisse di interpolazione
%
%Outputs:
% m: vettore dei coefficienti angolari
% r: vettore della costante di integrazione r
% q: vettore della costante di integrazione q
% h: vettore degli intervalli
%

function [m,r,q,h] = CalcolaSplineCubicaNaturale(x,f)

n = length(x);
if n~=length(f), error('Vettori non compatibili'), end

%Calcolo gli h(i)
for i = 1:n-1
    h(i) = x(i+1)-x(i);
end

%crea la matrice dei coefficienti
A = eye(n-2)*2;

%calcolo della prima colonna
A(2,1) = h(2)/(h(2)+h(3));

%calcolo delle colonne 2:(n-1)
for j = 2:n-3
        A(j-1,j) = h(j)/(h(j-1)+h(j));
        A(j+1,j) = h(j+1)/(h(j+2)+h(j+1));
```

```
end

%calcolo ultima colonna
A(n-3,n-2) = h(n-1)/(h(n-1)+h(n-2));

%calcolo differenze divise

dd = f;
for i = 1:2
    for j = n:-1:i+1
        dd(j) = (dd(j)-dd(j-1))/(x(j)-x(j-i));
    end
end
dd = dd(3:n)*6;


%risoluzione del sistema
[L,U] = unFattLU(fattLUtridiag(A));

y = sisem(L,dd','ibd');
m = sisem(U,y,'sbd');


%m va da 1 a n-2 con gli indici shiftati a sinistra di 1
%q ed r hanno l'indice shiftato di 1 a sinistra (partono da 1 e non da 2)
r(1) = f(1);
q(1) = (f(2)-f(1))/h(1) - (h(1)/6)*(m(1));



for i = 2:n-2
    r(i)= f(i)-((h(i)^2/6)*m(i-1));
    q(i)=(((f(i+1)-f(i))/h(i))-((h(i)/6)*(m(i)-m(i-1))));
end

r(n-1) = f(n-1)-(h(n-1)^2/6)*m(n-2);
q(n-1) = (f(n)-f(n-1))/h(n-1) - (h(n-1)/6)*(-m(n-2));
```

6.7.3 Spline cubica not-a-knot

Questi codici effettuano il calcolo e la valutazione in un array di punti di una spline cubica con condizioni not-a-knot:

```
%Questo metodo calcola e valuta una spline cubica not-a-knot
%
% x0 = ValutaSplineCubicaNAK(x,y,x0)
%
%Inputs:
% x: ascisse di interpolazione
% y: valori della funzione corrispondenti alle ascisse di interpolazione
% x0: vettore dei punti incui calcolare la spline
%
%Outputs:
```

```
% x0: vettore contenete il valore della spline sui punti
%     presi in input
%

function x0 = ValutaSplineCubicaNAK(x,y,x0)


[m,r,q,h] = CalcolaSplineCubicaNAK(x,y);

k = length(x0);
n = length(x);

for c = 1:k


         for i = 1:n-1
             if (((x0(c)) <= (x(i+1))) & ((x0(c)) >= (x(i))))
                x0(c) = r(i)+q(i)*(x0(c)-x(i))+( ((x0(c)-x(i))^3*m(i+1)...
                                        + (x(i+1)-x0(c))^3*m(i))/(6*h(i))  );
                break
             end

         end

end

%Questo metodo calcola  una spline cubica not-a-knot
%
% [m,r,q,h] = CalcolaSplineCubicaNAKno(x,f)
%
%Inputs:
% x: ascisse di interpolazione
% f: valori della funzione corrispondenti alle ascisse di interpolazione
%
%Outputs:
% m: vettore dei coefficienti angolari
% r: vettore della costante di integrazione r
% q: vettore della costante di integrazione q
% h: vettore degli intervalli
%


function [m,r,q,h] = CalcolaSplineCubicaNAK(x,f)

n = length(x);
if n~=length(f), error('Vettori non compatibili'), end
if n<4, error('Pochi punti di interpolazione'), end

%Calcolo gli h(i)
for i = 1:n-1
    h(i) = x(i+1)-x(i);
end

%crea la matrice dei coefficienti
```

```
A = eye(n)*2;

%phi e xi vano da 1 a n-2
for i = 1:n-2
phi(i) = h(i)/(h(i)+h(i+1));
xi(i) = h(i+1)/(h(i)+h(i+1));
end

%calcolo della prima e seconda colonna
A(1,1) = 1;
A(2,1) = phi(1);
A(2,2) = 2-phi(1);
A(2,3) = xi(1)-phi(1);

A(n,n) = 1;
A(n-1,n) = xi(n-2);
A(n-1,n-1) = 2-xi(n-2);
A(n-1,n-2) = phi(n-2)-xi(n-2);

for i = 3:n-2
A(i,i-1) = phi(i-1);
A(i,i+1) = xi(i-1);
end


%calcolo differenze divise

dd = f;
for i = 1:2
    for j = n:-1:i+1
        dd(j) = (dd(j)-dd(j-1))/(x(j)-x(j-i));
    end
end

dd(2:n-1) = dd(3:n)*6;

dd(1) = dd(2);
dd(n) = dd(n-1);


%risoluzione del sistema
[L,U] = unFattLU(fattLUtridiag(A));

y = sisem(L,dd',’ibd’);
m = sisem(U,y,’sbd’);


m(1) = m(1)-m(2)-m(3);
m(n) = m(n)-m(n-1)-m(n-2);


for i = 1:n-1
    r(i)= f(i)-((h(i)^2/6)*m(i));
    q(i)=(((f(i+1)-f(i))/h(i))-((h(i)/6)*(m(i+1)-m(i))));
```

```
end
```

6.7.4 Spline cubica completa

Questi codici effettuano il calcolo e la valutazione in un array di punti di una spline cubica Completa:

```
%Questo metodo calcola e valuta una spline cubica completa
%
% x0 = ValutaSplineCubicaCompleta(x,y,f1a,f1b,x0)
%
%Inputs:
% x: ascisse di interpolazione
% y: valori della funzione corrispondenti alle ascisse di interpolazione
% f1a: valore della derivata prima della funzione nell'estremo sinistro
% f1b: valore della derivata prima della funzione nell'estremo destro
% x0: vettore dei punti incui calcolare la spline
%
%Outputs:
% x0: vettore contenete il valore della spline sui punti
%     presi in input
%

function x0 = ValutaSplineCubicaCompleta(x,y,f1a,f1b,x0)


[m,r,q,h] = CalcolaSplineCubicaCompleta(x,y,f1a,f1b);

k = length(x0);
n = length(x);

for c = 1:k


         for i = 1:n-1
             if (((x0(c)) <= (x(i+1))) & ((x0(c)) >= (x(i))))
     x0(c) = r(i)+q(i)*(x0(c)-x(i))+( ((x0(c)-x(i))^3*m(i+1)...
                                          + (x(i+1)-x0(c))^3*m(i))/(6*h(i))  );
                break
             end

         end

end

%Questo metodo calcola  una spline cubica not-a-knot - versione non ottimizzata
%
% [m,r,q,h] = CalcolaSplineCubicaNAKno(x,f)
%
%Inputs:
% x: ascisse di interpolazione
% f: valori della funzione corrispondenti alle ascisse di interpolazione
% f1a: valore della derivata prima nell'estremo sinistro
```

```
% f1b: valore della derivata prima nell'estremo destro
%
%Outputs:
% m: vettore dei coefficienti angolari
% r: vettore della costante di integrazione r
% q: vettore della costante di integrazione q
% h: vettore degli intervalli
%


function [m,r,q,h] = CalcolaSplineCubicaCompleta(x,f,f1a,f1b)

n = length(x);
if n~=length(f), error('Vettori non compatibili'), end

%Calcolo gli h(i)
for i = 1:n-1
h(i) = x(i+1)-x(i);
end

%crea la matrice dei coefficienti
A = eye(n)*2;

%phi e xi vano da 1 a n-2
for i = 1:n-2
phi(i) = h(i)/(h(i)+h(i+1));
xi(i) = h(i+1)/(h(i)+h(i+1));
end

%calcolo della prima e seconda colonna
A(1,1) = -4*h(1);
A(1,2) = h(1);


%calcolo delle colonne 2:(n-1)
for j = 2:n-1
A(j,j-1) = phi(j-1);
A(j,j+1) = xi(j-1);
end

%calcolo ultima e penultima colonna
A(n,n) =4*h(n-1);
A(n,n-1) = -h(n-1);


%calcolo differenze divise
dd = f;
for i = 1:2
for j = n:-1:i+1
dd(j) = (dd(j)-dd(j-1))/(x(j)-x(j-i));
end
end

dd(2:n-1) = dd(3:n)*6;
```

```
dd(1) = 6*f1a-6*((f(2)-f(1))/(x(2)-x(1)));
dd(n) = 6*f1b-6*((f(n)-f(n-1))/(x(n)-x(n-1)));


%risoluzione del sistema
[L,U] = unFattLU(fattLUtridiag(A));

y = sisem(L,dd','ibd');
m = sisem(U,y,'sbd');

for i = 1:n-1
r(i)= f(i)-((h(i)^2/6)*m(i));
q(i)=(((f(i+1)-f(i))/h(i))-((h(i)/6)*(m(i+1)-m(i))));
end
```

6.7.5 Spline cubica periodica

Questi codici effettuano il calcolo e la valutazione in un array di punti di una spline cubica Periodica:

```
%Questo metodo calcola e valuta una spline cubica periodica
%
% x0 = ValutaSplineCubicaPeriodica(x,y,x0)
%
%Inputs:
% x: ascisse di interpolazione
% y: valori della funzione corrispondenti alle ascisse di interpolazione
% x0: vettore dei punti incui calcolare la spline
%
%Outputs:
% x0: vettore contenete il valore della spline sui punti
%     presi in input
%


function x0 = ValutaSplineCubicaPeriodica(x,y,x0)

[m,r,q,h] = CalcolaSplineCubicaPeriodica(x,y);

k = length(x0);
n = length(x);

for c = 1:k


         for i = 1:n-1
             if (((x0(c)) <= (x(i+1))) & ((x0(c)) >= (x(i))))
     x0(c) = r(i)+q(i)*(x0(c)-x(i))+( ((x0(c)-x(i))^3*m(i+1)...
                                        + (x(i+1)-x0(c))^3*m(i))/(6*h(i))  );
                break
             end

         end
```

```
end

%Questo metodo calcola  una spline cubica peropdica
%
% [m,r,q,h] = CalcolaSplineCubicaPeriodica(x,f)
%
%Inputs:
% x: ascisse di interpolazione
% f: valori della funzione corrispondenti alle ascisse di interpolazione
%
%Outputs:
% m: vettore dei coefficienti angolari
% r: vettore della costante di integrazione r
% q: vettore della costante di integrazione q
% h: vettore degli intervalli
%

function [m,r,q,h] = CalcolaSplineCubicaPeriodica(x,f)

n = length(x);
if n~=length(f), error('Vettori non compatibili'), end
if n<4, error('Pochi punti di interpolazione'), end

%Calcolo gli h(i)
for i = 1:n-1
h(i) = x(i+1)-x(i);
end

%crea la matrice dei coefficienti
A = eye(n)*2;

%phi e xi vano da 1 a n-2
for i = 1:n-2
phi(i) = h(i)/(h(i)+h(i+1));
xi(i) = h(i+1)/(h(i)+h(i+1));
end

%condizione m_1=m_n
A(1,1) = 1;
A(1,n) = -1;


%calcolo delle colonne 2:(n-1)
for j = 2:n-1
A(j,j-1) = phi(j-1);
A(j,j+1) = xi(j-1);
end

%conizione s'(a)=s'(b)
A(n,1) = -2*h(1);
A(n,2) = -h(1);
A(n,n-1) = -h(n-1);
A(n,n) = -2*h(n-1);
```

```
%calcolo differenze divise

dd = f;
for i = 1:2
for j = n:-1:i+1
dd(j) = (dd(j)-dd(j-1))/(x(j)-x(j-i));
end
end

dd(2:n-1) = dd(3:n)*6;
dd(1) = 0;
dd(n) = 6*(((f(n)-f(n-1))/(x(n)-x(n-1)))-((f(2)-f(1))/(x(2)-x(1))));


%risoluzione del sistema

%[L,U] = unFattLU(fattLUtridiag(A));

%y = sisem(L,dd',’ibd’);
%m = sisem(U,y,’sbd’);

m = solveLUpivot(A,dd’);


for i = 1:n-1
r(i)= f(i)-((h(i)^2/6)*m(i));
q(i)=(((f(i+1)-f(i))/h(i))-((h(i)/6)*(m(i+1)-m(i))));
end
```

6.7.6 Approssimazione ai mini quadrati

Qusto codice calcola e valuta il polinomio che risolve il problema dei minimi quadrati:

```
%Calcola il polinomio interpolante di migliore approssimazione ai minimi quadrati
%
%           ff = ValutaPolMinimiQuadrati(x,f,g,x0)
%
%Questo metodo prende in input:
%     x: ascisse di interpolazione
%     f: valori della funzione corrispondenti alle ascisse di interpolazione
%     g: grado del polinomio desiderato
%     x0: vettore di punti in cui valutare il polinomio
%     D: matrice dei pesi, mettere 0 se non utilizzata.
%e restituisce
%     ff: array contenente la valutazione del polinomio nei punti x0.
function ff = ValutaPolMinimiQuadrati(x,f,g,x0,D)


[m,n] = size(f);
[i,j] = size(x);
if (m~=i)or(n~=j), error('I vettori devono avere la stessa dimensione'), end
```

```
if n>m, f= f';x=x';m=n;, end

%Creazione della matrice

A(:,g+1) = ones(m,1);
for j = 1:g
for i = 1:m
A(i,j) = x(i)^(g-j+1);
end
end
if D~=0
      f = D*f;
      A = D*A;
end;

ff = solveQR(A,f)';

ff = Horner(ff,x0);
```

6.8 Formule di quadratura

6.8.1 Formula dei trapezi adattiva non ricorsiva

```
%formula adattiva dei trapezi non ricorsiva
%
%     [I,p] = TrapeziAdattivaNRfop(f,a,b,tol)
%
%Questo metodo prende in input
%     f: funzione integranda
%     a: estremo sinistro dell'intervallo di integrazione
%     b: estremo destro dell'intervallo di integrazione
%     tol: tolleranza sul risultato
%e restituisce:
%     I: valore dell'integrale
%     p: i punti di valutazione della funzione
function [I,p] = TrapeziAdattivaNRfop(f,a,b,tol)


p = [a,b, tol;];
go(1) = 1;
k=0;
while(max(go)==1 & k<10000)
k=k+1;
for i = 1:size(p,1)
if go(i) == 1
if p(i,1)~=p(i,2)
I1 = NewtonCotes(f,p(i,1),p(i,2),1);
                        I2(i) = NewtonCotes(f,p(i,1),((p(i,1)+p(i,2))/2),1)...
                                +NewtonCotes(f,((p(i,1)+p(i,2))/2),p(i,2),1);
else, I1=0; I2(i)=0;, end
if abs(I1-I2(i))/3<=p(i,3)
go(i) = 0;
l = size(p,1);
```

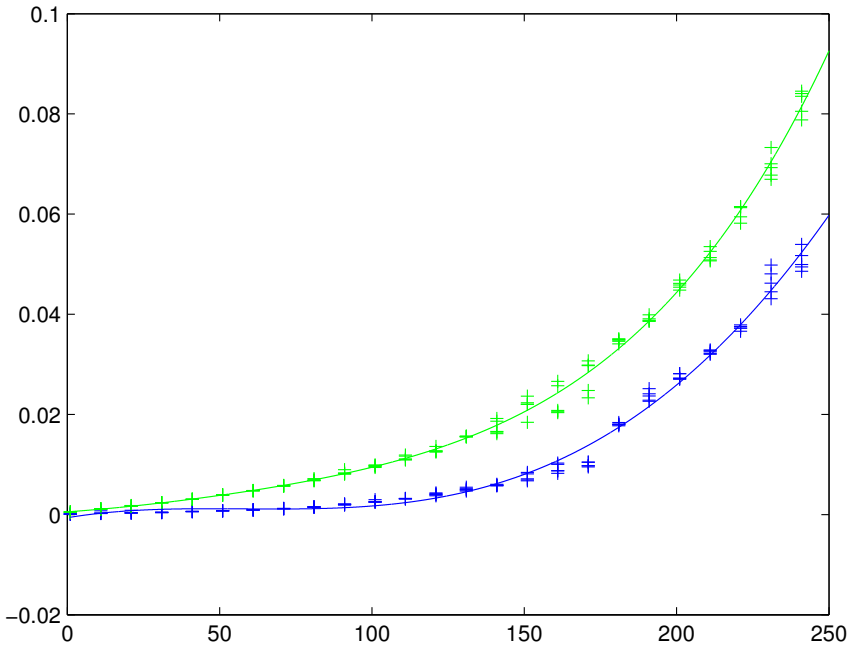

Figura 6.5: Ecco un esempio di approssimazione ai minimi quadrati di una serie di dati sperimentali: in questo grafico stiamo approssimando i dati relativi ai tempi di fattorizzazione LDL^T relativi al metodo fattLDLlib (blu) e vectorizeLDL (verde).

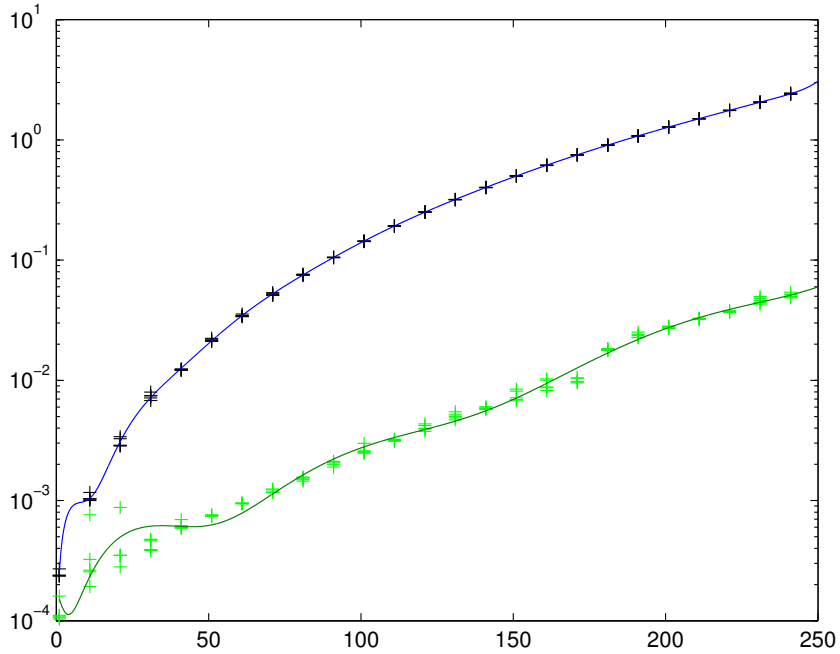

Figura 6.6: Approssimazione ai minimi quadrati dei dati relativi al metodo fattLDL (blu) e fattLDLlib (verde).

```
go(l+1) = 0;
p(l+1,2) = p(i,2);
p(i,2) =  ((p(i,1)+p(i,2))/2);
p(l+1,1) = p(i,2);

else
go(i) = 1;
l = size(p,1);
go(l+1) = 1;
p(l+1,2) = p(i,2);
p(i,2) =  ((p(i,1)+p(i,2))/2);
p(l+1,1) = p(i,2);
p(i,3) = p(i,3)/2;
p(l+1,3) = p(i,3);

end
end
end
end

p = [sort(p(:,1));max(p(:,2))];
I = sum(I2);
```

6.8.2 Formula di Simpson adattiva non ricorsiva

```
%formula adattiva di Simpson non ricorsiva
%
%     [I,p] = SimpsonAdattivaNRfop(f,a,b,tol)
%
%Questo metodo prende in input
%     f: funzione integranda
%     a: estremo sinistro dell'intervallo di integrazione
%     b: estremo destro dell'intervallo di integrazione
%     tol: tolleranza sul risultato
%e restituisce:
%     I: valore dell'integrale
%     p: i punti di valutazione della funzione
function [I,p] = SimpsonAdattivaNRfop(f,a,b,tol)

p = [a,b, tol;];
go(1,1) = 1;
k = 0;
I = 0;
I2 = 0;
while(max(go)==1 & k<1000)
k=k+1;
for i = 1:size(p,1)
if go(i) == 1
if p(i,1)~=p(i,2)
                                %I1 = NewtonCotesComp(f,p(i,1),p(i,2),2,2)
                                I2 = NewtonCotesComp(f,p(i,1),p(i,2),2,4);
I1 = NewtonCotes(f,p(i,1),p(i,2),2);
%I2 = NewtonCotes(f,p(i,1),((p(i,1)...
                    %+p(i,2))/2),2)+NewtonCotes(f,((p(i,1)+p(i,2))/2),p(i,2),2)
```

```
else, I1=0; I2=0;, end
if (abs(I1-I2)/15)<=p(i,3)
I = I+I2;
a = p(i,1);
b = p(i,2);
s = (b-a)/4;
l = size(p,1);
p(i,1) = a;
p(i,2) =a+s;
p(l+1,1) = p(i,2);
p(l+1,2) = a+2*s;
p(l+2,1) = a+2*s;
p(l+2,2) = a+3*s;
p(l+3,1) = a+3*s;
p(l+3,2) = b;
go(i) = 0;
go(l+1) = 0;
go(l+2) = 0;
go(l+3) = 0;
else
a = p(i,1);
b = p(i,2);
go(i) = 1;
l = size(p,1);
go(l+1) = 1;
p(l+1,2) = b;
p(i,2) =  (a+b)/2;
p(l+1,1) = (a+b)/2;
p(i,3) = p(i,3)/2;
  p(l+1,3) = p(i,3);
end
end
end

end
p = [sort(p(:,1));max(p(:,2))];
```